`PSysCal`: Disseny i implementació d'un sistema paral·lel per al calibratge automàtic de models P-Sistema

Treball de Final de Carrera

Enginyeria en Informàtica

Escola Politècnica Superior

Universitat de Lleida

Autor:
Albert Agraz Sánchez

Directors:
Josep Lluís Lérida Monsó i Francesc Solsona Tehàs

Juny de 2012

Agraïments

M'agradaria donar les gràcies a moltes persones que m'han ajudat en el decurs dels meus anys a la Universitat. Des de professors a amics i companys de classe. No puc nombrar-los aquí a tots, però doneu-vos per al·ludits.

En la realització d'aquest projecte, ha estat imprescindible l'ajuda prestada per els dos tutors Josep Lluís Lérida i Francesc Solsona. De la mateixa forma, no hagués estat possible sense la col·laboració de na Maria Àngels Colomer.

Més enllà d'aquests, també voldria remarcar la referència i el suport moral que ha estat per mi en Francisco Clarià. La seva guia ha estat de gran ajuda.

Finalment, i no només per aquest projecte, voldria agrair a la meva família el seu suport incondicional, i les estones de patiment que han compartit amb mi; sempre hi són quan els necessites, d'una forma o altra, guiant els meus passos.

Resum

Els `P-Systems` són un tipus de models computacionals basats en el funcionament intern de les cèl·lules biològiques que ens permeten simular sistemes amb incertesa de forma pareguda als processos naturals.
Actualment, aquests models tenen una gran aplicació en el camp de la modelització d'ecosistemes ja que, com s'ha demostrat, aporten uns resultats molt satisfactoris.

Existeixen diverses implementacions d'aquest tipus de models, però destaca sobre la resta el llenguatge `P-Lingua`[2] i el motor de simulació `P-Lingua Core`[1], desenvolupat per un equip d'investigació de la Universitat de Sevilla.

Moltes vegades però, es fa difícil determinar els valors exactes de certs paràmetres que influeixen en el model, degut a que la captura de dades de camp no és sempre fidedigna, o bé senzillament no s'han pres les mesures adequades en el moment determinat.
Aquest fet fa que, un cop definit correctament el model, s'hagin de realitzar diverses proves fins a trobar el valor correcte d'aquests paràmetres. D'aquest procés se'n diu calibratge, i és molt important ja que marcarà la diferència entre un model vàlid i un que no ho és.

Actualment aquest procés es realitza de forma manual i seguint l'"instint" dels investigadors de camp. En aquest projecte proposem un sistema per al calibratge automàtic de models `P-System` que explori tots els possibles valors dels paràmetres i triï el que produeix uns resultats més semblants a la realitat.

El problema que es planteja és que el nombre de combinacions de possibles valors dels paràmetres és un problema d'explosió combinatòria, i implica una gran quantitat de simulacions a realitzar per tal de trobar-ne la millor. La solució proposada en aquest projecte és la realització d'aquestes simulacions de forma paral·lela.

Presentarem doncs, una aplicació sèrie que permeti el calibratge automàtic de models `P-System` de mida reduïda, i una aplicació paral·lela que ens permeti atacar models amb una complexitat superior. Hem comprovat que l'aplicació paral·lela pot reduir el temps d'execució un $95,57\%$ respecte l'aplicació sèrie.
A més a més, aquesta aplicació ha demostrat tenir un *Speedup* sempre molt proper al nombre de nodes, mantenint una eficiència superior al 99% entre 1 i 12 nodes i només perdent un $2,2\%$ en el cas de 24 nodes.

Amb tot això, podem afirmar que l'aplicació paral·lela presentada és una aplicació altament eficient i escalable, ja que manté aquesta eficiència al llarg de les proves amb un nombre de nodes creixent.

Tot i haver presentat una aplicació paral·lela que soluciona aparentment els problemes de temps de l'aplicació sèrie, encara presenta certes limitacions en quant a la mida del problema, així, per sistemes amb molts paràmetres a calibrar i rangs amplis d'aquests paràmetres encara presenten certes limitacions. Per a solucionar aquest problema, s'indiquen noves vies d'investigació que involucren tècniques d'Intel·ligència Artificial.

Índex

Índex de figures

Índex de taules

Índex d'Algoritmes

Índex de Codi Font

Capítol 1

Introducció

La simulació matemàtica de processos naturals és de gran utilitat per a poder conèixer els seus mecanismes de funcionament i poder predir el seu comportament sota diferents escenaris plausibles. Per això, el modelatge de processos és una forma molt utilitzada per tal d'avançar en els àmbits de recerca aplicada.

La metodologia de modelat ha evolucionat a mesura que el coneixement de la realitat ha anat augmentant. Les prestacions dels sistemes informàtics també han augmentat amb la constant evolució de la indústria tecnològica. Això ha portat a desenvolupar metodologies de modelat molt completes, basades en la gran capacitat de còmput disponible.

Avui en dia, podem deixar d'associar el concepte de modelatge a l'existència de complexes equacions matemàtiques que relacionen, amb més o menys èxit, unes entrades amb unes sortides. En l'actualitat, s'ha produït una gran expansió en l'ús de models computacionals, dels que val la pena destacar els models de viabilitat i els multi-agents.
L'última generació de models són força més senzills d'entendre, però també força més difícils de plantejar, ja que no es tracta d'una representació a partir de fórmules analítiques.

El modelat de sistemes s'ha aplicat satisfactòriament a multitud de sistemes; des del modelatge de les condicions climatològiques terrestres, fins a la simulació del plegament de proteïnes, passant per la codificació de les cadenes d'ADN.

En concret, nosaltres treballarem amb un tipus de models computacionals que són els `P-Systems`. Aquesta tècnica de modelatge està inspirada en el funcionament intern d'una cèl·lula biològica; amb els seus orgànuls i la seva membrana.
Aquest tipus de model es pot aplicar a qualsevol tipus d'ecosistema, realitzant les adaptacions pertinents. S'han utilitzat aquests `P-Systems` amb resultats més que satisfactoris en l'estudi del trencalòs dels Pirineus [4][5][6][7], del mol·lusc zebra a l'embassament de Ribarroja o del tritó pirinenc [8], que utilitzarem en aquest projecte com a model de referència.

Els ingredients bàsics d'un `P-System` són 3: l'estructura jeràrquica de membranes, el conjunt de regles que s'apliquen per a evolucionar el model i l'alfabet d'objectes que es troben en les membranes. Les membranes representen els conjunts dins del model. Les regles són els patrons d'evolució del sistema, i els objectes són els elements que trobem en el model.

Els investigadors, al realitzar un model, defineixen les regles i les membranes

a partir de l'observació del medi. L'alfabet d'objectes també es defineix de la mateixa forma, però en aquest cas han de determinar els valors inicials de certs paràmetres. Aquests valors acostumen a correspondre a característiques del medi prèviament observades. De totes formes, hi ha certes característiques que són molt difícils d'observar (per exemple, el nombre de descendents d'una espècie que moren abans dels 3 mesos) o que no s'han observat en el moment oportú. Per a sortejar aquesta dificultat i poder assignar un valor inicial correcte a tots els paràmetres, s'acostuma a confiar en l'experiència dels investigadors de camp per a donar un valor per aquests. De totes formes, s'han de realitzar diverses proves i mirar a veure si aquest valor realment s'ajusta a la realitat; en definitiva, s'ha de calibrar el model.

Actualment, aquest calibratge del model es porta a terme a mà i seguint una mica l'"instint" de l'investigador, que serà qui determinarà -a través de l'assaig/error- el valor més adient per al paràmetre en qüestió. El que nosaltres proposem en aquest projecte és un sistema automàtic per al calibratge de models `P-Systems`.

A continuació explicarem quines són les eines emprades en aquest projecte i més endavant l'estructura d'aquesta memòria.

1.1 Eines emprades

En el desenvolupament d'aquest projecte utilitzarem un llenguatge de representació de `P-Systems` anomenat `P-Lingua`. Aquest llenguatge ha estat desenvolupat per investigadors de la Universidad de Sevilla dins del marc d'un projecte amb el mateix nom [2].

Aquest grup de recerca ha desenvolupat de la mateixa forma un motor de simulació anomenat `P-Lingua Core`[3] que accepta models realitzats amb `P-Lingua`, i en simula la seva evolució. Aquest motor s'ofereix com una llibreria `Java` que es pot executar important-la o bé com una aplicació independent.

En quant al llenguatge de programació de la nostra pròpia aplicació, hem volgut desmarcar-nos una mica de les típiques opcions que es veuen en les aplicacions amb orientació paral·lela i ens hem decantat per `Python`. Aquest llenguatge ofereix un gran repertori de funcions per recórrer llistes i treballar de forma molt eficient amb conjunts de dades.
En el cas que ens ocupa, hem de tractar amb un gran volum de dades i és per aquest motiu que l'hem escollit.

1.2 Estructura d'aquesta memòria

Hem estructurat aquesta memòria intentant plasmar el procés seguit en el transcurs del projecte. Un cop finalitzada la introducció, analitzarem el procés de modelatge de sistemes que segueixen els investigadors en el seu dia a dia, per tal d'entendre amb quina mena de processos estem treballant.

El següent pas és analitzar el procés de calibratge de sistemes, i els motius que fan que, fins ara, no s'hagi adoptat un sistema que ho faci de forma automàtica. Acte seguit proposarem un procés per a realitzar aquest calibratge automàtic.

Havent explicat ja la proposta, arriba el moment de dissenyar i implementar la primera versió d'aquesta aplicació. Es tractarà de la versió sèrie del sistema. Explicarem amb detall alguns dels conflictes que hem trobat a l'hora de fer-ho.

Degut al gran requeriment de temps de l'aplicació sèrie, desenvoluparem una aplicació paral·lela que realitzi la mateixa tasca però en un temps molt menor. Primer haurem d'analitzar quines àrees són paral·lelitzables, i com ho podríem fer. Un cop decidida l'arquitectura d'aquesta aplicació, explicarem el seu disseny així com el paradigma de programació paral·lel/distribuït escollit.

Finalment, realitzarem els experiments per tal de validar la solució proposada. Un cop presentats els resultats serà el moment d'extreure les conclusions del treball realitzat i indicar el treball futur.

Trobarem també uns apèndixs amb informació complementària. El primer apèndix conté l'article presentat a les XXIII Jornadas de Paralelismo organitzat per SARTECO. El segon i tercer apèndixs contenen el codi de les aplicacions sèrie i paral·lela, respectivament. El quart apèndix mostra el model utilitzat per a realitzar les proves d'aquest projecte i, finalment, l'últim apèndix conté un estudi preliminar que es va realitzar per a trobar una llibreria eficient de `MPI` per a `Python`.

Capítol 2

Modelatge de sistemes

El primer que hem de fer per entendre el projecte dut a terme és aprofundir en una sèrie de conceptes clau per al seu desenvolupament. Ja hem vist que aquest projecte tracta sobre el calibratge de models computacionals realitzats amb `P-Systems`.

La simulació de processos naturals té un especial interès per a poder predir el seu comportament sota diferents situacions plausibles, per tal de prendre decisions sobre el medi.

La metodologia de modelatge ha evolucionat a mesura que la ciència ha avançat i ha fet que els conexiements sobre la realitat hagin augmentat. A mes a més, la impressionant evolució dels sistemes informàtics, han posat a disposició dels científics grans capacitats de còmput i emmagatzemament. Aquests fets han canviat enormement la forma d'abordar els problemes i les eines de modelatge que s'utilitzen.

A continuació farem un estudi amb més profunditat dels `P-Systems` per a poder entendre completament què implica l'ús d'aquests sistemes i perquè són una bona eina per a simular processos naturals. Després explicarem el procés que segueixen els investigadors per a crear models de sistemes.

2.1 Tipus de models

Un model d'ecosistema és una representació abstracta, habitualment matemàtica o a partir de regles lògiques, que reprodueix un sistema ecològic.

Existeixen diversos tipus de models d'ecosistemes disponibles en la literatura actual. Aquests tipus s'acostumen a classificar en dos grans classes: models analítics o models computacionals.

Els models analítics són models a partir de complexes equacions matemàtiques que defineixen molt precisament el comportament de processos estudiats de forma exhaustiva. Acostumen a ser utilitzats per a explicar sistemes relativament senzills; d'alta forma, les equacions es converteixen en intractables.

Els models computacionals (també coneguts com models de simulació) es basen en tècniques numèriques per a representar els processos que es duen a terme en el si d'un ecosistema [9]. Disposen de regles força més senzilles, i es basen en la simulació amb grans quantitats de dades.

En la pràctica, realitzar models computacionals és més senzill, i són força

més utilitzats ja que permeten representar més fidelment la naturalesa intrínseca dels processos naturals, que disten de ser tant exactes com les expressions matemàtiques dels models analítics.

Una de les propietats que és desitjable que tinguin aquests models computacionals que siguin fàcilment executables. En els casos de simulació d'ecosistemes, també es busca la possibilitat de reproduir o representar l'atzar propi dels ecosistemes. Això es pot aconseguir de diverses formes. Un dels tipus de models computacionals que millor capturen aquesta característica són els `P-Systems`.

Aquest ha estat el tipus de model escollit per a la realització d'aquest projecte. No ha estat una decisió presa al llarg de la realització del mateix, sinó que era el tipus de modelatge que s'estava utilitzant amb anterioritat a la iniciació d'aquest projecte per part dels usuaris ecòlegs.

A continuació analitzarem en detall els `P-Systems` i els components que els formen.

2.2 P-Systems

Els `P-Systems` són models computacionals per a la simulació de processos inspirats en el funcionament intern de les cèl·lules que trobem en els organismes. Van ser introduïts al 1998 pel matemàtic George Păun [10][11][12].

Les cèl·lules són organismes diminuts amb unes zones diferenciades en el seu interior, que contenen una sèrie d'orgànuls que operen de forma simultània i aleatòria; comunicant-se i intercanviant materials amb el mitjà que les rodeja.

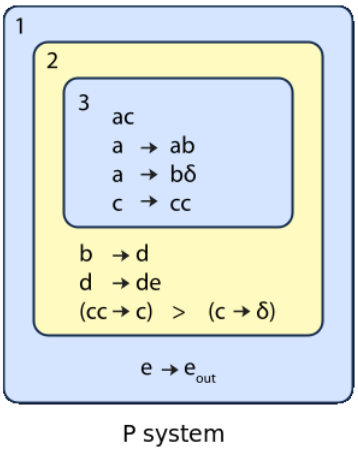

Figura 2.1: Esquema d'un `P-System`.

Els ingredients bàsics d'un `P-System` els podem trobar en la figura 2.1 són:

- **Entorn**: es tracta de l'embolcall dels `P-Systems`. Acostumen a estar formats per més d'un `P-System`, de tal forma que s'entenen els entorns com zones on interactuen diversos ecosistemes.

- **Membranes**: són les principals estructures dins del `P-System`. Es tracta d'una unitat amb càrrega positiva (+) o negativa (-) que pot contenir altres

membranes. L'espai entre aquestes membranes pot contenir objectes. Les regles s'apliquen sobre les membranes, per tant, també sobre els elements que aquestes contenen, ja sigui objectes, com altres membranes.
Representen les "fronteres"que trobem dins d'una cèl·lula, que delimiten els orgànuls que trobem dins d'aquestes.

- **Objectes**: en l'interior de cada membrana trobem un conjunt d'objectes (que pot ser buit, per tant, no tenir-ne) que representen els continguts químics dels orgànuls que formen la cèl·lula. En l'aplicació dels `P-Systems` a altres entorns, es tradueixen els objectes als recursos, o elements que trobem en cada zona. En el cas de representació d'ecosistemes, podrien ser els recursos disponibles, etc.
  Trobem dos tipus d'objectes. Els objectes tradicionals s'operen durant el transcurs de les regles. Els catalitzadors, però, són necessaris per a certes regles per a succeir, però no es consumeixen fruit de l'aplicació d'aquestes regles.

- **Regles**: les regles són les construccions semàntiques mínimes que representen els possibles canvis que sofreix el sistema. Cada regla té una sèrie de requeriments per a que es pugui aplicar, per exemple, un determinat nombre d'objectes amb una cardinalitat mínima present. També es poden determinar prioritats entre regles, definint que certa regla s'executarà, a igualtat de disponibilitat, abans que una altra.
  Les regles s'apliquen sobre una membrana i, per tant, sobre els elements d'aquesta. La selecció de quina regla (sempre i que tinguin la mateixa prioritat) aplicar, és on s'introdueix l'atzar necessari per a reproduir millor els processos naturals.
  Veurem la forma de les regles més endavant.

A banda d'aquests ingredients descrits, és molt important per a aquest projecte parlar dels **paràmetres** del model. Els paràmetres són valors que els ecòlegs defineixen i que representen l'estat del medi. Aquests paràmetres hauran de tenir un valor per a poder començar la simulació del model.
Per a qualsevol model donat, hi haurà paràmetres dels que es coneixerà el seu valor perquè s'ha pogut observar a l'ecosistema, però també hi haurà paràmetres dels que desconeixem el valor, ja sigui perquè aquest valor no s'ha pogut observar en el medi en el moment adequat o senzillament perquè no és observable.
Aquests paràmetres són representats com objectes dins de la definició del model, però són dues entitats diferents.

La simulació d'un model consisteix en l'aplicació recursiva de les regles fins arribar a una configuració final. És a dir, a partir d'una configuració inicial de les membranes i els seus objectes, aplicarem totes les possibles regles. Quan no hagi més regles aplicables, haurem avançat fins a la següent configuració. Tornarem a aplicar les regles i avançarem una altra configuració. Efectuarem aquesta operació successivament fins arribar a un punt en que ja hagem evolucionat al llarg d'un nombre determinat de configuracions (definit per l'usuari) o bé no disposem de més regles a aplicar.
Els models dels ecosistemes no disposen d'un final. No tindria sentit que un ecosistema ben representat s'acabés en un punt en concret. Per això indicarem

el nombre de configuracions que volem evolucionar per tal de parar la simulació en algun moment.

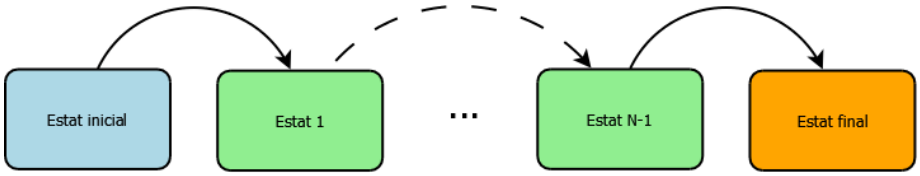

Figura 2.2: Evolució de la simulació d'un `P-System`.

En essència, la simulació d'un model es defineix com una successió de configuracions que comença en la configuració inicial i acaba en l'estat final (veure figura 2.2). Cada estat es forma per l'estructura en aquell moment de les membranes junt amb els objectes i els seus valors que es troben dins de cada membrana.

En el següent apartat veurem els diferents variants de models de `P-Systems` que trobem i les diferències i similituds entre ells.

Variants de `P-Systems`

Existeixen diferents variants de `P-Systems` que formen diferents tipus de models. Estudiant la bibliografia [13][14], trobem els següents casos:

- Sistemes de Transició: són els primers creats per George Păun com a model bàsic de computació amb membranes. La seva característica principal és que utilitzen les regles del model com a transició entre estats durant l'evolució del sistema. La resta de variants han estat evolucions d'aquesta, per tal d'afegir més opcions o solucionar mancances.

- Sistemes amb Membranes Actives: són una variant introduïda l'any 2000 per el mateix George Păun per tal d'atacar amb computació de membranes els problemes NP-complets. La incorporació principal és l'opció de dividir membranes (no confondre amb dissolució) en membranes més petites. Així, hi haurà més tipus de regles que seran aplicables a aquests sistemes: evolució, comunicació, dissolució i divisió.
  Aquesta variant, però presenta alguna altra diferència, conseqüència de les noves característiques: no s'utilitzen prioritats, les regles s'apliquen simultàniament, primer s'aplicarà la regla d'evolució, i després la de dissolució, la membrana més exterior no es pot dissoldre ni dividir, ...

- Sistemes Probabilístics: es tracta de l'evolució natural que indica que en el cas de trobar una situació amb dues o més regles de possible aplicació s'escollirà una d'elles amb la probabilitat P, enlloc d'executar les dues.

- Sistemes Multientorn: en els sistemes biològics, s'acostumen a formar sistemes complexos a través de la congregació de sistemes més petits. Els sistemes multientorn permeten aquest funcionament, definint diversos entorns en cada model. Els entorns es podran comunicar entre ells.

Com hem vist, a partir del `P-System` més senzill introduït per George Păun a l'inici de la seva investigació sobre computació per membranes, s'han anat evolucionant diverses variants que aporten noves capacitats als models i permeten representar realitats més complexes. Es tracta d'una ciència en evolució, i s'espera que en el futur vagin apareixent noves variants amb més opcions per als investigadors.

A continuació veurem quina es la sintaxi detallada de les regles, per a poder entendre millor el funcionament de la simulació.

Sintaxis de les regles

Les regles defineixen una transició. Aixi, en els models `P-Systems` trobem regles que es poden aplicar a les membranes i regles que es poden aplicar entre entorns.

Les regles que es poden aplica entre membranes són de la forma:

$$r \equiv u^a[v^b]_i^\alpha \xrightarrow{f_r} u'^c[v'^d]_i^{\alpha'}$$

Si trobem un objecte u amb multiplicitat a a l'exterior de la membrana i amb càrrega α y, en la membrana etiquetada com i, trobem l'objecte v amb multiplicitat b; amb una probabilitat f_r s'aplica la regla. Conseqüentment, canvia la càrrega de la membrana de α a α' i els objectes u i v evolucionen a objectes u' i v' amb multiplicitats c i d respectivament.

Les regles que s'apliquen entre entorns, en canvi, tenen la forma:

$$r_e \equiv (x)_{e_j} \xrightarrow{p(x,j,j_1,\cdots,j_k)} (y_1)_{e_{j_1}} \cdots (y_k)_{e_{j_k}}$$

L'objecte x passa de l'entorn e_j als entorns $e_{j_1} \cdots e_{j_k}$. En aquest pas entre entorns, podrà evolucionar als objectes respectius $y_1 \cdots y_k$.

El següent pas és veure un senzill exemple d'execució.

Exemple d'execució

En la figura 2.1 podem observar l'estat inicial d'un `P-System`. La membrana exterior (1) és l'entorn d'aquest sistema, i conté la regla de `sortida`. La segona membrana conté 4 regles (dos d'elles en una relació de prioritat), la última de les quals conté el símbol especial δ que representa l'ordre de dissolució de la membrana. La membrana més interior conté un conjunt d'objectes ac i 3 regles. Cal remarcar que la única membrana que disposa d'objectes en l'estat inicial és la 3, tot i que, com veurem al llarg de la simulació, aquests aniran movent-se.

L'atzar inherent als processos naturals i correctament capturat per aquest tipus de modelatge, fa que no hi hagi una única evolució possible d'un escenari. A continuació presentem una possible simulació:

1. De l'estat inicial, només la membrana 3 disposa dels objectes "ac". Així:
   - S'aplica la regla $c \to cc$ sobre c.
   - S'aplica la regla $a \to ab$ sobre a. Aquí es podria haver executat alternativament la regla $a \to a\delta$, però per l'atzar s'ha escollit la primera.

   Al finalitzar aquest pas, disposem de $abcc$ a la membrana 3.

2. Aplicarem ara les regles de la membrana 3 sobre els seus objectes:

   - S'aplica al regla $a \to a\delta$, tot i que també es podria haver aplicat la regla $a \to ab$.
   - S'aplica la regla $c \to cc$ sobre els dos elements c de la membrana.

   Ara, el contingut de la membrana és $bb\delta cccc$. Al trobar l'objecte α es dissol la membrana, i els objectes passen directament a la membrana exterior. D'aquesta manera, la membrana 2 contindrà els elements $bbcccc$.

3. Ara tocarà aplicar les regles de la membrana 2 als objectes:

   - S'aplica la regla $b \to d$ a les dues b que hi ha.
   - S'aplica la regla $cc \to c$ a les 2 aparicions de cc.

   Finalitzarem aquest pas amb els objectes $ddcc$ a la membrana 2.

4. Continuarem aplicant les regles de la membrana 2 als objectes $ddcc$:

   - S'aplica la regla $d \to de$ a les dues aparicions de d.
   - S'aplica la regla $cc \to c$ a cc.

   Al finalitzar aquesta aplicació, tindrem $dedec$ a la membrana 2.

5. Seguim aplicant les regles a la membrana 2:

   - S'aplica $d \to de$ dues vegades.
   - S'aplica $c \to \delta$.

   Trobem el símbol δ, per tant, dissoldrem la membrana 2.

6. Ara la membrana 1 conté $deedee$:

   - S'aplica 4 vegades la regla $e \to e_{out}$.

   Ara tindrem $de_{out}e_{out}de_{out}e_{out}$ a la membrana 1.

7. Donat que la única regla que hi ha a la membrana 1 s'aplica sobre e i aquest objecte no es troba en el contingut d'aquesta membrana, no podem aplicar cap regla. Ha arribat el moment d'**aturar la simulació**.

Un cop ja hem vist què és un `P-System`, és el moment d'analitzar el procés de modelatge d'ecosistemes que segueixen els investigadors a l'hora de definir un model.

2.3 Procés de modelatge

És molt important veure quin és el procediment actual que segueixen els investigadors a l'hora de crear els models per als sistemes que estudien. Entendre correctament aquest procés, ens permetrà apreciar la magnitud del problema que tenim entre mans així com la necessitat de l'aplicació que estem creant.

De forma global, el procés es pot resumir en la figura 2.3. En aquesta figura veiem com està basat en tres fases: disseny, calibratge i aplicació. Més endavant veurem amb detall cada una d'aquestes fases.

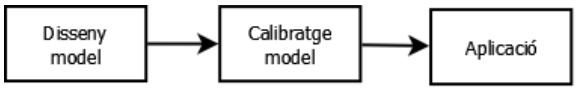

Figura 2.3: Procés general de modelatge.

El primer que farem és establir un llenguatge comú. Per fer-ho, farem diverses definicions de conceptes que ens acompanyaran al llarg d'aquest document. Després estudiarem el procediment de creació de models pas a pas.

2.3.1 Definicions

Com ja hem vist, un model està format per un entorn, l'estructura de membranes, un alfabet d'objectes i un conjunt de regles. Definim un **model** com el conjunt de l'entorn, l'estructura de membranes, l'alfabet d'objectes i les regles. En concret, en un model, l'alfabet d'objectes no tenen un valor inicial assignat.

Si assignem un valor als **paràmetres** del model, aleshores estarem parlant d'un **escenari**. D'aquesta manera, per cada model tenim tants escenaris com valors inicials a les variables podem assignar. En un sentit estricte de la definició, estaríem parlant d'infinits escenaris, però tenint en compte els significats físics de les variables, aquestes només prendran valors plausibles en la realitat. Així, aquestes variables poden prendre tots els valors possibles dins d'un rang de realitat, que acotarem amb un pas per tal de convertir en discret.

En efecte, l'estat inicial que hem estat tractant fins ara, es tracta d'un possible escenari del model en qüestió.

2.3.2 Anàlisi del procés

Ara que ja disposem d'un llenguatge comú, és el moment de veure com els investigadors creen un model i troben l'escenari adient d'aquest.

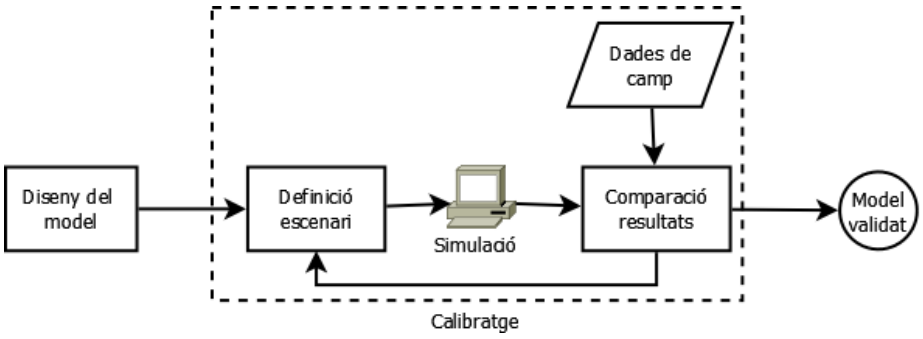

Figura 2.4: Proces de modelatge de `P-Systems`.

Com veiem en la figura 2.4, el procés de modelatge s'inicia amb el **disseny del model**. Aquest pas representa l'estudi de l'ecosistema per a definir aquelles regles de funcionament que observem. També es definiran en aquest cas totes les membranes i la estructura que tindran. A més a més definirem els objectes

que es trobaran en aquestes membranes. Específicament no definirem els valors dels **paràmetres** del model.

La definició dels valors inicials dels paràmetres (o variables) d'aquest model portarà a la **determinació d'un escenari**. Per trobar els valors d'aquestes variables (o paràmetres) es donen dues situacions: poden ser valors concrets i coneguts o bé podem desconèixer-ne el valor exacte.

En el cas de conèixer el valor, ja sigui perquè és un valor fixe (per exemple, el coeficient de gravetat) o bé perquè l'hem pogut observar directament en el camp de treball, assignarem aquest valor a la variable i quedarà fixat definitivament en l'escenari.
Tots els futurs escenaris possibles compartiran aquests valors, ja que no donen lloc a cap mena de dubte de la seva validesa. Si un escenari falla en ser validat, no podrà ser per motiu d'aquests valors ja que han sigut capturats de la realitat.

L'altra situació és que no coneguem aquests valors. Podem desconèixer el valor d'una variable ja sigui perquè no s'han pres les dades de camp necessàries per determinar-lo o bé perquè és "humanamentïmpossible capturar aquest valor a la realitat (per exemple, el volum de fulles caigudes en un període concret de temps en un bosc en concret). Fins i tot es pot donar la situació que siguin valors que canvien entre anys, i que la seva correcta observació de forma successiva doni resultats diferents.

En els casos en que no disposem d'un valor determinat (o bé no tinguem un valor únic per al paràmetre), ens assessorament per els investigadors de camp per tal de definir un valor de forma arbitrària. Aquest valor no pot prendre qualsevol forma, sinó que està dins d'un rang de valors possibles físicament. Així, no tindria sentit un índex d'humitat relativa alt en un desert, o un 0% d'humitat relativa a ran de mar.

La definició correcta dels valors dels paràmetres és cabdal per aconseguir un model fidel a la realitat. S'ha observat que una mala definició dels valors inicials pot invalidar completament un model ben definit, així, caldrà trobar un escenari correcte per a que el model sigui eficaç. Com veurem, es dedicaran molts esforços a trobar aquest escenari.

El següent pas en el procés de modelatge és **simular** l'escenari. El que farà l'investigador és llençar la simulació de l'escenari per un període de temps determinat. Degut a l'atzar propi de la variant dels models `P-Systems` utilitzada, l'investigador simularà l'escenari entre 30 i 100 vegades i promitjarà els resultats per tal de minimitzar l'efecte d'aquest atzar en els resultats.

Hem de remarcar que aquesta simulació no és una operació lleugera de fer. Si considerem que a més a més, es realitzarà un mínim de 30 vegades, resulta un cost computacional i de temps força alt. Estem parlant, en un cas típic, d'execucions de 90 segons. Així,

$$90 \text{ segons} \cdot 30 \text{ repeticions} = 45 \text{ minuts}$$

$$90 \text{ segons} \cdot 100 \text{ repeticions} = 2h30 \text{ minuts}$$

Simularem cada escenari per un període de temps que ja coneixem, és a dir, per un període de temps que ja ha passat. En aquest sentit, per exemple, establirem l'escenari inicial amb les dades de l'ecosistema de fa 10 anys i farem una simulació de 10 anys. Si el model és correcte, ens retornarà l'estat actual de l'ecosistema.
Aquesta idea es pot aplicar en qualsevol període de temps, sempre i quant

les condicions generals de l'ecosistema (les regles o les membranes) no hagin canviat en la realitat. Així, podríem establir l'escenari inicial amb les dades de l'any 1920 i simular 5 anys. Aleshores les dades haurien de coincidir amb les obtingudes de l'observació de camp de l'any 1925.
És important escollir un període de temps del que coneguem amb exactitud el màxim nombre de dades inicials i que finalitzi en un moment on puguem determinar (o ja hagem determinat) els valors dels objectes més rellevants per al model (per exemple, censos de població, etc.). La diferència és que els valors inicials acostumen a ser força més (i més difícils de capturar) que els resultats.

Un cop simulat l'escenari i obtinguts els resultats finals, haurem de determinar si aquest escenari és vàlid o no. Per fer-ho, com ja hem comentat, **comparerem els resultats** obtinguts amb les dades de camp capturades de la realitat. Establirem un llindar d'error admissible, i qualsevol escenari que tingui un error inferior a aquest serà acceptat com a vàlid. Parlem d'un llindar i no exigim una correlació exacta de les dades perquè som conscient de que estem treballant amb dades de camp, i aquestes sempre porten un error intrínsec.

Quan parlem d'error ens acostumem a referir a la distància euclidiana entre els resultats obtinguts per la simulació i les dades de camp. Aquesta definició, però, no és la única possible, ja que trobem moltes altres opcions per a determinar errors en cossos multivariables. De totes formes, en aquest projecte treballarem amb aquesta distància.

En el cas de que l'escenari no sigui validat (comet un error superior al llindar) retornarem al segon pas. En aquesta situació haurem de modificar els valors inicials d'aquelles variables que han estat definides de forma arbitrària, és a dir, haurem d'escollir un altre escenari per a simular.
Com sembla natural, no modificarem el valor inicial d'aquells paràmetres que sí que vam poder observar directament a l'ecosistema, ja que estem segurs que aquests valors són els que han estat assignats.

Val a dir, que un escenari podria ser invàlid per que les regles que defineix no són vàlides. Aquests casos, però, no es donen amb tanta freqüència com els de l'error en el valor inicial dels paràmetres i acostumen a ser detectats en la fase de disseny del model.

Realitzarem aquest bucle tantes vegades com sigui necessari, perfilant els valors inicials d'aquelles variables incertes sempre dins del rang -i mai perdent de vista- d'allò físicament possible. Si arribem a explorar totes les possibles combinacions de valors d'aquests elements i no hem trobat un escenari que sigui capaç de mantenir un error per sota del llindar, determinarem que el model és invàlid.

2.4 Necessitat de calibratge

Com ja hem vist, en el procés de modelatge existeix una fase molt important que cerca un escenari que sigui suficientment fidel a la realitat. Anomenarem aquesta fase **calibratge** del sistema. Aquesta fase és vital ja que una bona realització de la mateixa pot significar la diferència entre un model validat o bé un model rebutjat.

Ara per ara, aquest procés no es realitza de forma sistemàtica, sinó que es fa d'una forma arbitrària, seguint les intuïcions dels experts en el camp estudiat. Moltes vegades, aquesta forma de fer el calibratge porta a una pèrdua de temps

significativa, i a desesperar en el intent.

A més a més, fer-ho d'una forma arbitrària fa que puguem arribar a trobar un escenari suficientment bo (que generi un error per sota del llindar) però que no sigui el millor escenari possible, amb el que no estarem explotant tota la potència i definició del model.

Tot això fa palès que es necessari una aplicació que ens permeti realitzar aquest calibratge d'una forma automàtica i desatesa. D'aquesta manera es pot executar aquesta aplicació i esperar com a resultat el millor escenari possible per al model donat. Evidentment, aquesta aplicació només hauria de considerar aquells valors de les variables indeterminades que siguin físicament possibles.

A continuació, veurem una proposta d'un sistema automàtic de calibratge que doni solucions a aquestes exigències.

Capítol 3

Calibratge de sistemes

Com hem vist en la secció 2.4, la creació d'una aplicació que permeti realitzar el calibratge automàtic dels models, trobant aquell escenari que cometés el mínim error possible a l'hora de simular els processos que representa significaria una gran avenç en el procés de disseny de models.

Si analitzem les contrapartides que pot arribar a presentar, ho entendrem. En el següent apartat analitzarem els problemes que presenta aquesta forma de calibratge (en contraposició al calibratge discret o arbitrari). Més endavant proposarem una solució per al problema.

3.1 Problemes del calibratge

Essencialment el que volem fer és generar tots els possibles escenaris i avaluar-los. Aleshores calcular l'error que comet cada escenari i escollir el millor. Si aquest escenari comet un error inferior al llindar definit per l'usuari, el model quedarà validat amb aquest escenari. Per contra, si no ho aconsegueix, el model quedarà invalidat.

Com ja hem vist, hi haurà un gran nombre de paràmetres per als que ja coneixem el seu valor inicial a partir de l'observació directa del medi. En canvi, hi haurà un altre conjunt menor de paràmetres que l'usuari voldrà ajustar. Serà per aquells valors que no hem pogut capturar que realitzarem el calibratge.

Així doncs, haurem de permetre a l'usuari definir quins són aquests paràmetres i establir un rang de valors en que aquests paràmetres són vàlids. Aquest rang serà determinat per allò que és físicament possible en l'entorn estudiant. Per exemple, l'índex de precipitacions del Pirineu, no podrà arribar a l'índex que trobarem a les zones tropicals en època de monsons.

Un cop definits els paràmetres a calibrar, haurem de provar totes les possibles combinacions amb tots els valors d'aquests paràmetres. Això significa que fixarem tots els paràmetres a un valor determinat menys un, aleshores provarem tots els possibles valors d'aquest últim, mantenint el valor dels altres. Quan hagem provat tots aquests valors, canviarem un punt en un altre element, i tornarem a recórrer tots els valors del primer paràmetre.

Si observem el problema, es tracta d'un problema d'**explosió combinatòria**, és a dir, si augmentem lleugerament el rang a calibrar d'un paràmetre, el nombre d'escenaris a provar augmenta de forma exponencial.

Tenint en compte el gran cost de simular cada escenari, l'explosió combinatòria representa un gran problema a l'hora d'obtenir els resultats en un temps raonable.

En un cas típic, estarem parlant d'un nombre aproximat de 800 escenaris a provar. En aquest cas, i recuperant les dades de l'apartat anterior,

$$800 \text{ escenaris} \cdot 45 \frac{\text{minuts}}{\text{escenari}} = 25 \text{ dies}$$

Com veiem, pot arribar a representar un problema disposar d'un únic equip monolític destinat a executar aquest còmput.

Un altre problema derivat del gran nombre d'escenaris a avaluar és l'**espai de disc** necessari per a emmagatzemar la ingent quantitat de dades que es produeixen, tant per generar l'escenari a resultat de la seva simulació.

En els estudis realitzats, s'han obtingut valors que oscil·len entre $0,5$GB per a un escenari amb 30 repeticions als $1,6$GB en cas de demanar 100 repeticions. Tenint en compte una execució de 800 escenaris:

$$800 \text{ escenaris} \cdot 0,5 \frac{\text{GB}}{\text{escenari}} = 400\text{GB}$$

$$800 \text{ escenaris} \cdot 1,6 \frac{\text{GB}}{\text{escenari}} = 1,2\text{TB}$$

Per a realitzar aquestes proves s'ha hagut d'instal·lar un servidor de disc dedicat, ja que les instal·lacions de la Universitat no suportaven aquesta càrrega de dades.

3.2 Procés de calibratge automàtic

Un cop vista la necessitat de calibratge cal buscar solucions que permetin ajustar els models en un temps raonable i de forma automàtica. En aquesta secció presentem el procediment general a seguir, i més endavant detallarem com implementem aquest procediment.

Realitzarem un tipus de calibratge exhaustiu, és a dir, explorarem tots els possibles escenaris del model per tal de trobar-ne el "millor". Considerarem el "millor"escenari aquell que aconsegueixi uns resultats més propers a la realitat.

El que volem és automatitzar el procés de calibratge, és a dir, que es converteixi en un procés completament desatès, el final del qual sigui el model una vegada calibrat, és a dir, el millor escenari.

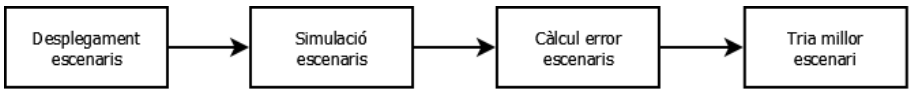

Figura 3.1: Procés de calibratge automàtic.

Com mostra la figura 3.1, el nostre procés es basa en 4 passos. El primer pas és el **desplegament d'escenaris**. Aquest desplegament significa calcular tots els possibles escenaris a partir del model i dels possibles valors que pot prendre cada paràmetre per al que el valor inicial no està determinat.

És important reflexionar en aquest cas sobre la necessitat d'emmagatzemament que ha de tenir el nostre dispositiu. Tot i que crear els escenaris no és

tant costós en quant a espai de disc com la seva simulació, també requereix una quantitat d'espai considerable. Més endavant veurem una possible solució a aquest problema.

El següent pas és la **simulació dels escenaris**. Consisteix en simular cada escenari un nombre determinat de vegades (entre 30 i 100) i després calcular el promig dels resultats normalitzats per cada escenari.

En aquest cas sí que s'ha de tenir molt en compte l'espai de disc ja que, com hem vist en la secció anterior, les necessitats d'emmagatzematge poden arribar a ser prohibitives per qualsevol sistema de sobretaula. Més endavant, veurem una opció per a solucionar aquest problema.

Un cop simulats els escenaris i promitjats els resultats, és el moment del **càlcul de l'error**. Compararem els resultats amb la realitat i obtindrem un valor que representarà l'error de l'escenari; és a dir, quant diferent és de la realitat.

Finalment, escollirem el millor escenari d'entre tots els que hem simulat. Per a fer això, mirarem un a un i sempre ens quedarem amb el mínim. Per implementar el càlcul de l'error podem pensar en una solució Master-Worker en el que el Master s'encarregaria de processar els errors un cop obtingudes totes les solucions, o bé que cada Worker faci el càlcul de l'error de l'escenari processat de manera que el Master un cops processats tots els escenaris nomes hauria de consultar quin és l'escenari amb l'error mínim. En el nostre treball, hem presentat una solució que contempla la segona d'aquestes opcions.

Solució al problema de l'emmagatzemament

Com ja hem vist, existeix un problema amb la quantitat d'espai d'emmagatzemament a utilitzar. També podem observar, que cada un dels passos que hem descrit en la figura 3.1 representa recórrer tots els possibles escenaris.

Se'ns acudeix canviar la forma d'executar el algoritme, i en lloc de centrarnos en les operacions i executar-les sobre tots els escenaris, ens podem centrar en recórrer tots els escenaris i executar totes les operacions per cada un d'ells. El canvi el podem veure en l'algoritme 1.

Algorithm 1 Procés de calibratge.

```
1: escenaris := desplegar_escenaris();
2: for escenari in escenaris do
3:    fitxer := generar_fitxer_escenari(escenari);
4:    for i in N do
5:       resultats[i] := simular(fitxer);
6:    end for
7:    resultat := promitjar(resultats);
8:    error := distancia_euclidea(resultat,real);
9:    if error < error_min then
10:      error_min := error;
11:      escenari_min := escenari;
12:   end if
13:   esborra_fitxers_escenari(escenari);
14: end for
15: return escenari
```

Aquesta millora calcula primer tots els possibles escenaris (línia 1) però no genera els fitxers per cada escenari. Enlloc d'això entra en un bucle que recorre cada un dels escenaris (línies 2-14) i per cada un dels escenaris:

1. Genera el fitxer de l'escenari i l'escriu al disc (línia 3).
2. Simula l'escenari un nombre N de vegades (línies de 4 a 6).
3. Calcula el promitg dels resultats d'aquestes repeticions (línia 7).
4. Calcula l'error a partir del resultat del promitg i de les dades de la realitat (línia 8).
5. Comprova si és l'escenari amb error mínim fins al moment (línies de 9 a 12).
6. Esborra els fitxers generats fins al moment per aquell escenari (línia 13). Això inclou el fitxer de l'escenari i tots els fitxers de resultats de les simulacions de les repeticions.

Amb aquest procediment s'observa que podem treballar cada escenari de forma independent de la resta i que no és necessari generar tots els escenaris, sinó que en cada pas de l'algoritme nomes necessitem les dades de l'escenari actiu, reduint molt les necessitats d'espai de disc.

Capítol 4

Aplicació sèrie

En l'apartat anterior hem vist una proposta de solució al problema del calibratge automàtic. Hem presentat, fins i tot, un possible algoritme per aquesta implementació. Ara és el moment de veure com hem materialitzat aquesta idea en una aplicació real.

Per a la implementació d'aquesta aplicació hem utilitzat el llenguatge de programació `Python`[15]. La tria d'aquest llenguatge s'ha dut a terme per la facilitat que aquest aporta a l'hora de recòrrer conjunts de dades i estructures. Hem valorat que es tracta d'un llenguatge interpretat força ràpid i que ens permetrà treballar molt còmodament en l'exploració exhaustiva d'escenaris.

Es tracta d'un llenguatge força popular (entre els 10 més utilitzats arreu del món), que aporta una forma de programar força senzilla en relació amb la sintaxis natural que utilitzem. Permetrà l'ús successiu d'operacions modulars de tal forma que resulta molt senzill llegir i entendre el codi.

Evidentment, es tracta d'un llenguatge interpretat que pot ser més lent que la referència de C, tot i així, la gran riquesa en les estructures de dades, la facilitat en la seva gestió i la potencia d'aquestes funcions fan que `Python` sigui una alternativa més que interessant a considerar en la implementació d'aplicacions per a ser executades en entorns de clústers distribuïts. S'han vist últimament diverses implementacions d'algoritmes de gran consum de còmput amb `Python`, amb molts bons resultats de rendiment.

Existeixen diverses planes web que donen indicacions per a implementar codi en `Python` d'una forma eficient [16]. Nosaltres hem valorat molt positivament la senzillesa de la programació i la immediatesa de l'obtenció de resultats en el procés de codificació. Considerem que aquestes característiques, en un entorn d'investigació són molt importants.

A més a més, es tracta d'un llenguatge sobre el que no s'han desenvolupat gaires aplicacions paral·leles.

A continuació estudiarem dos aspectes molt importants en el desenvolupament de l'aplicació: la identificació d'escenaris i els fitxers d'entrada al sistema. Un cop definits aquestes dues especificacions, analitzarem l'algoritme implementat de forma detallada.

El codi en `Python` d'aquesta aplicació sèrie s'adjunta en l'apèndix B.

4.1 Identificació d'Escenaris

Una de les decisions que hem hagut de prendre al llarg del desenvolupament d'aquest projecte ha estat la forma de definir els escenaris. Recordem que el conjunt de regles amb els objectes (sense valor assignat) forma el model, mentre que si assignem valor a aquestes objectes estem davant d'un escenari.

Hi haurà un nombre finit d'escenaris per cada model, que dependran del nombre de paràmetres d'aquest model per al que desconeixem un valor inicial i del rang de possibles valors que aquestes poden prendre. Es tracta d'un problema combinatori com hem exposat amb anterioritat.

Necessitarem un sistema d'identificació dels escenaris segons els valors dels paràmetres a calibrar, per tal de poder identificar els resultats per cada escenari, o bé per poder generar nous escenaris durant el procés de calibratge.

El problema se'ns planteja en la forma d'identificar cada escenari de forma unívoca. Una forma de fer-ho seria identificar-los a través de la llista de variables per calibrar i del seu valor concret en aquest escenari. Això planteja que un possible identificador seria: `m{3,1}=0.05, m{4,1}=0.05, m{5,1}=0.05, p{1}=0.55, p{2}=0.05`.

Aquesta opció representa un problema en el sentit que utilitzar això com un identificador dins de l'aplicació informàtica podrà representar un problema en el moment de comparació, etc. A més a més, ens deixa diverses incògnites sobre si l'ordre de les variables és rellevant, si els valors poden prendre valors amb 4 decimals o bé només permet un nombre fixat, etc.

Per tots aquests motius, hem dissenyat un sistema que permeti identificar els escenaris de forma més senzilla, i amb un cost computacional molt reduït. Abans de veure aquest sistema en la pràctica, hem d'introduir el concepte de *taula de calibratge*.

Taula 4.1: Taula de calibratge.

Nombre	Valores					
m{3,1}	0.05	0.06	0.07	0.08	0.09	0.10
m{4,1}	0.05	0.07	0.09	0.11	0.13	
m{5,1}	0.05	0.06	0.07	0.08	0.09	0.10
p{1}	0.55	0.60	0.65			
p{2}	0.05	0.06	0.07	0.08	0.09	0.10

Una **taula de calibratge** (taula 4.1) és la representació de totes les variables a calibrar amb els seus possibles valors. Les files representaran les variables, mentre que les columnes seran els possibles valors. Com sembla evident, cada fila podrà tenir un nombre de columnes diferents, que dependrà de la mida del rang de possibles valors que l'expert de camp hagi definit per aquella variable en concret.

Reconstruir aquesta taula a partir d'un fitxer de text definit per l'expert de camp consumeix un temps despreciable amb `Python`. A més a més podrem identificar una combinació de valors pels índexs de cada fila (paràmetre), això fa que sigui molt fàcil i ràpid identificar els valors dels paràmetres donat un conjunt d'índexs i, per tant, és una opció força interessant per implementar la identificació d'escenaris.

L'avantatge que ens aporta l'ús d'aquesta taula és que, de bones a primeres, assigna un ordre a les variables. Així, no queda a l'atzar l'ordenació de les mateixes en una cadena de text. A més a més, de cara a la precisió dels nombres, només tindrem la limitació que ens assigni el llenguatge de programació, ja que es tracta d'una taula en memòria i no en una expressió de text.

Basats en aquesta taula, proposem un sistema per identificar els escenaris que funciona com segueix. El que farem és definir un vector amb tantes posicions com variables a calibrar, es a dir, tantes files com hi ha a la taula. En tot moment mantindrem l'ordre de les variables en la taula. A continuació assignarem a la posició de cada variable, el nombre de columna (començant per zero) on es troba el valor assignat a aquella variable per a l'escenari en concret.

Per exemple, en el cas de la taula 4.1, l'escenari amb valors

$$\mathtt{m}\{3,1\} = 0.05, \mathtt{m}\{4,1\} = 0.05, \mathtt{m}\{5,1\} = 0.05, \mathtt{p}\{1\} = 0.55, \mathtt{p}\{2\} = 0.55$$

tindrà com identificador el vector `(0,0,0,0,0)`, mentre que l'escenari

$$\mathtt{m}\{3,1\} = 0.08, \mathtt{m}\{4,1\} = 0.07, \mathtt{m}\{5,1\} = 0.07, \mathtt{p}\{1\} = 0.55, \mathtt{p}\{2\} = 0.10$$

tindrà per identificador el vector `(3,1,2,0,5)`.

Aquest sistema és que ens permetrà identificar de forma senzilla els escenaris, per molts que n'hi hagi.

A l'hora de recórrer els escenaris d'aquesta forma, ho podem fer de dues formes: considerant que el bit menys significatiu es troba a l'esquerra o a la dreta. En els dos casos, l'escenari inicial és el `(0,0,0,0,0)`, però següent escenari ja canvia: serà `(1,0,0,0,0)` si el bit menys significatiu es troba a l'esquerra i `(0,0,0,0,1)` si es troba a la dreta.

Per passar d'un escenari al següent, però, no és suficient amb sumar una unitat al bit menys significatiu, sinó que s'ha de comprovar si aquest fet provoca que aquest índex sigui superior al rang de possibles valors per aquella variable (és a dir, si intentem accedir a una columna que no existeix).

Hem de dissenyar, doncs, una petita funció que ens serveixi per a trobar el següent escenari a partir d'un escenari base. Com a convenció, utilitzarem escenaris amb el bit menys significatiu a l'esquerra. Tot el que seguirà és perfectament convertible per els escenaris amb el bit menys significatiu a la dreta; tant sols s'ha de canviar l'ordre de recórrer els índexs, de i a 0 enlloc de 0 a i.

La idea d'aquest algoritme és augmentar sempre en una unitat el bit menys significatiu, i aleshores iniciar un bucle per totes les posicions (de la menys significativa a la més) per tal de mirar si aquest augment d'una unitat ha fet que sobrepassem el nombre de valors a provar per el paràmetre concret.

En cas de trobar aquesta situació, ficarem a zero la posició que s'ha excedit i augmentarem en una unitat la posició del següent paràmetre. El mateix bucle ens portarà a analitzar, aquest cop, la posició que hem augmentat, així que anem propagant l'excés al llarg de totes les posicions, de la menys significativa a la més.

Situarem diverses instruccions de trencament del bucle per no haver de recórrer totes les posicions sempre. De fet, a la que trobem una posició que no ha sobrepassat el seu límit, no caldrà que mirem les següents, ja que no serà possible que aquestes s'excedeixin si no reben la suma de la posició anterior.

L'algoritme 2 ens mostra com hem implementat aquesta operació. Pas a pas, el primer que fa l'algoritme 2 és augmentar en una unitat l'índex de la posició

Algorithm 2 Obtenció del següent escenari.

```
1:  it[0] := it[0] + 1;
2:  for i in nombre_variables do
3:    if it[k] ⩾ length(valors_variables[k]) then
4:      it[k] := 0;
5:      if k = ultima_variables ∧ it[k+1] ⩾ length(valors_variables[k]) then
6:        it[k+1] := it[k+1] + 1;
7:      else
8:        break;
9:      end if
10:   else
11:     break;
12:   end if
13: end for
```

menys significativa del vector (línia 1). Aleshores, per cada posició del vector (línies de la 2 a la 12), realitzarà les següents operacions:

1. Comprova si en augmentar una unitat l'índex de la posició actual es sobrepassa el rang de possibles valors per aquesta variable. En cas de no sobrepassar-lo, surt del bucle i acaba la funció (línies 10 i 11).

2. En cas de superar l'índex màxim, fica el valor de la posició actual a zero. Aleshores,

   - Si és l'últim valor de la posició més significativa, surt del bucle i acaba; ja que hem arribat al final (línia 8).
   - Si no, augmenta en una unitat l'índex de la següent posició (línia 6) i analitza la següent posició del vector (a la que hem sumat 1).

Hem realitzat diversos estudis sobre quines estructures de dades utilitzar en la seva implementació, per tal de no perdre eficiència en l'ús de `Python` (veure apèndix B). Hem dedicat força treball a la seva optimització ja que s'executarà aquesta funció tantes vegades com combinacions estiguem analitzant d'un model, es a dir, moltes vegades.

4.2 Fitxers d'entrada i sortida

L'entrada i la sortida del nostre sistema ens permetrà copsar millor com treballa. Conèixer quins paràmetres captura el nostre sistema i quines són les respostes del mateix ens ajudarà a veure quin efecte tindrà en el sistema.

Com podem veure en la figura 4.1, els fitxers que entraran en el sistema són els següents:

- **Fitxer de Regles**: es tracta d'un fitxer `.pli` amb totes les regles que defineixen el model. Aquest fitxer el realitzaran els ecòlegs involucrats en el procés de creació del model. Les regles es definiran com s'ha especificat en l'apartat 2.2 d'aquesta memòria.

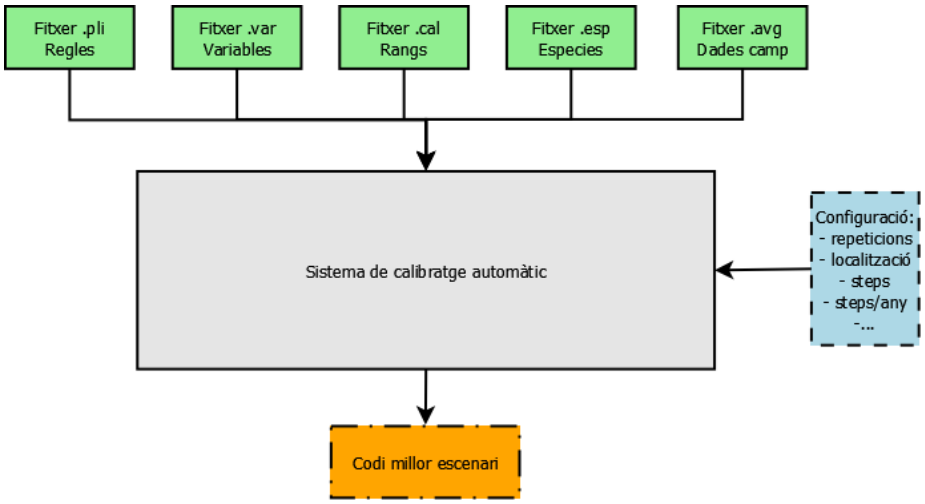

Figura 4.1: Esquema d'entrada i sortida de dades.

- **Fitxer de Variables**: en aquest fitxer trobarem totes les variables que formen l'alfabet d'objectes, tant aquelles variables per les que tenim un valor inicial fixe, com aquelles que s'han d'ajustar.
  Indicarem el valor inicial per cada cas, tot i que en les variables que s'han de calibrar, ignorarem aquest valor, que serà substituït pels pertanyents al rang de calibratge.
  En aquest cas, el fitxer tindrà una terminació `.var`.

- **Fitxer de Rangs**: serà la definició d'aquells objectes de l'alfabet de variables que voldrem calibrar, junt amb els valors del rang.
  La terminació d'aquest fitxer serà `.cal` i la seva estructura serà d'una fila per cada variable a calibrar, amb el següent contingut:

  – Valor inicial
  – Valor final
  – Pas

  On *pas* l'utilitzarem per a generar els valors entre el *valor inicial* i el *valor final* (ambdós inclusius).
  Una fila d'exemple podria ser:

  ```
  q{1}=0.50:0.70:0.05
  ```

  On $q\{1\}$ serà la variable a calibrar, 0.50 el valor inicial, 0.70 el valor final i 0.05 el pas.

- **Fitxer d'Espècies**: en aquest document indicarem quines són les variables que volem estudiar, es a dir, aquelles que normalitzarem i compararem amb les dades de camp. Inclourem una fila per cada variable, i només hi indicarem el nom, per exemple:

```
q{1}
```

La terminació serà `.esp`.

- **Fitxer de Camp**: per a realitzar la valoració de l'efectivitat de cada escenari, necessitem un fitxer que ens indiqui quines són les dades reals, és a dir, les dades de camp. Compararem aquest fitxer amb els resultats dels escenaris per a conèixer l'error comés. Aquest fitxer tindrà una terminació `.avg` i l'estructura de cada fila serà:

  ```
  1:q{1}*17234.68
  ```

  Que significarà que a l'any 1, la variable $q\{1\}$ té un valor de 17234.68.

A més a més, al sistema haurem d'entrar alguns valors de **configuració**. Essencialment necessitarem el nombre de repeticions a simular per escenari, la localització del directori de treball, el nombre de passos que s'ha d'evolucionar l'ecosistema en cada simulació, la relació entre el nombre de passos i els anys, etc.

La **sortida del sistema** serà el codi de l'escenari que produeixi els resultats més propers a la realitat. Serà de la forma $(x_1, x_2, x_3, \ldots, x_i)$, i es retornarà per pantalla al finalitzar l'execució de l'aplicació sèrie. A partir d'aquest codi és fàcil per l'ecòleg identificar els valors d'ajust del model.

El procediment que seguiran els usuaris d'aquesta aplicació serà el de dissenyar un model, especificar els rangs de valors dels paràmetres a calibrar i aplicar-li el calibratge automàtic per a trobar el millor escenari, és a dir, els valors dels paràmetres que facin que el model produeixi resultats més pròxims a la realitat.
Un cop calibrat el model, aquest es podrà fer servir per la presa de decisions als gestors del medi, ...

4.3 Procés de calibratge

Definits ja les dades necessàries per al calibratge automàtic de models realitzats amb `P-Systems`, és el moment de presentar l'algoritme dissenyat per a implementar aquesta solució en un codi sèrie. Trobarem aquesta representació en l'algoritme 3.

És important remarcar que es tracta d'una versió més aplicada a la implementació que la proposada en l'algoritme 1. En aquest presentem una seqüència detallada d'operacions.

Com podem veure en l'algoritme 3, es requerirà l'**entrada** de tots els fitxers descrits en l'apartat 4.2 i de la configuració de l'aplicació. El primer que farem, doncs, és inicialitzar aquesta configuració en els paràmetres adequats de l'aplicació i llegirem els fitxers per a guardar-ne el contingut en memòria (línies 1-2). Amb els fitxers en memòria, reconstruirem la **taula de calibratge**, i inicialitzarem els **iteradors** (línies 3-4).
Els iteradors són unes estructures que ens permetran tenir en cada moment controlat quin valor dins del rang de cada variable estem estudiant. No emmagatzemaran el seu valor, sinó la posició d'aquest dins del rang de valors possibles. En tindrem, per tant, un per variable a calibrar.

Algorithm 3 Aplicació sèrie.

Require: Configuració i fitxers `.pli`, `.var`, `.cal`, `.esp` i `.avg`.

```
1: inicialitzar_configuracio();
2: llegir_fitxers();
3: taula := reconstruir_taula_calibratge();
4: it := inicialitzar_iteradors(taula);
5: nombre_escenaris := calcular_nombre_escenaris(it);
6: for (i := 0; i < nombre_escenaris; i++) do
7:    id_escenari := generar_id_escenari(it);
8:    fitxer_escenari := generar_fitxer_escenari(it);
9:    for i in N do
10:      resultats[i] := simular(fitxer);
11:   end for
12:   resultat := promitjar(resultats);
13:   error := distancia_euclidea(resultat,real);
14:   if error < error_min then
15:      error_min := error;
16:      escenari_min := escenari;
17:   end if
18:   esborrar_fitxers_escenari(escenari);
19:   calcular_seguent_escenari(it);
20: end for
21: return escenari_min
```

La *posició inicial* dels iteradors (fruit de l'aplicació de la línia 4) és tots a 0. Així, el primer escenari a estudiar sempre serà el `(0,0,...,0)`, i ja quedarà així preparat abans d'entrar en el bucle per a recórrer tots els escenaris.

En aquest punt és el moment de definir la **condició d'aturada** del programa. Aquesta condició serà un cop hem arribat a explorar el nombre total d'escenaris que es poden generar a partir de la taula de calibratge. Consisteix en multiplicar la cardinalitat de tots els rangs (línia 5). Aleshores iniciarem un bucle que donarà exactament tantes voltes com escenaris podem crear (línia 6).

Per cada escenari, el primer que farem és generar el seu **identificador** a partir de la posició actual dels iteradors (línia 7). Essencialment, i com ja hem descrit en detall, es tractarà de les posicions actuals de cada iterador amb l'ordre concret de les variables: `(3,1,4,...,0)`, per exemple.

A continuació es genera el **fitxer de l'escenari** a partir dels valors corresponents a les posicions actuals dels iteradors (línia 8). La primera secció del fitxer de l'escenari serà el contingut íntegre del fitxer `.pli` (les regles). La segona secció serà el llistat de variables amb el seu valor inicial, és a dir, el contingut del fitxer `.var`. En aquesta segona secció, però, haurem de substituïr els valors de les variables a calibrar per els corresponents a l'escenari actual. Això ho farem consultant en la taula de calibratge la posició indicada per l'iterador de cada variable.

Amb el fitxer de l'escenari generat, ja podrem entrar en un bucle per a **simular l'escenari** tantes vegades com s'hagi indicat en la configuració (línies de 9 1 11). El fet de guardar els resultats de cada simulació és un tema una mica complex. Com ja hem vist, el resultat de cada simulació és un fitxer amb la seqüència de

estats seguida per l'evolució i els valors de les variables en cada estat. El que farem és guardar tots aquests fitxers en un directori concret, per després poder trobar-los fàcilment. En l'algoritme, però, ho representem com si ho guardèssim en un vector.
Un cop simulades totes les repeticions, es farà necessari **promitjar**-les per tal de normalitzar els resultats (línia 12). Per això emprarem una llibreria desenvolupada a tal fi. Aquesta llibreria llegirà tots els fitxers de resultats, n'extraurà les dades rellevants i en farà un promitg de les dades (espècies) definides per l'usuari ecòleg.
Ara ja estem en disposició de realitzar la **comparació** de l'únic fitxer de resultats amb el fitxer de dades reals (línia 13). Altre cop, ens basarem en la llibreria esmentada per a realitzar aquesta comparació, i obtenir un valor únic d'error.
El següent pas a realitzar és comprovar si l'**error** obtingut en aquest últim escenari és inferior al mínim acumulat. En cas de ser-ho substituirem l'acumulat per l'últim escenari simulat (línies de 14 a 17).
L'últim pas rellevant en quant a l'escenari actual és el d'**esborrar els fitxers** relacionats amb aquest. Com ja hem comentat, es tracta d'una forma d'alleugerir la càrrega d'emmagatzemament sobre el sistema (línia 18).
Finalment, **calcularem el següent escenari** (línia 19). Això significa trobar la següent posició dels iteradors vàlida segons la taula de calibratge, tal i com hem descrit a l'algoritme 2.

En acabar totes les voltes necessàries calculades en la línia 5, el bucle acabarà **retornant** el codi de l'escenari que ha representat un error mínim en comparació amb les dades de camp entrades. Podem trobar una implementació d'aquest algoritme a l'apèndix B.

4.4 Mancances de l'aplicació sèrie

Un cop implementada l'aplicació sèrie, es realitzen proves sobre models d'ecosistemes reals, i s'obtenen els valors de temps per al calibratge automàtic. En aquest cas, ens interessa estudiar el temps que es tarda en dur a terme aquest.

Trobem, en l'anàlisi d'aquests resultats, una principal mancança: el temps. L'execució del nostre sistema de calibratge automàtic ens portarà molt temps. No estem parlant de dues hores, sinó un **mínim d'un dia** sencer per a ecosistemes senzills. Podem veure els resultats en l'apartat 6.

A més a més, segons els informes dels ecòlegs que utilitzaran el sistema, si aquest fos capaç de retornar resultats en un període de temps molt inferior, es plantejarien augmentar el nombre de variables a calibrar ja que, pel que sembla, hi ha certs casos en que les dades no són 100% fiables, tot i que s'accepten com a bones.

El problema temporal causat pel problema combinatori i la necessitat d'escalar el problema de calibratge per part dels usuaris ecòlegs fa que sigui imprescindible buscar una solució a aquesta implementació sèrie, que permeti resoldre el problema amb temps molt més assequible.

En els següents apartats proposem una solució paral·lela per al calibratge automàtic i analitzem el seu comportament sota un cas real.

Capítol 5

Paral·lelització de l'Aplicació

Com ja hem vist en l'apartat 4.4, la versió sèrie del sistema de calibratge automàtic presenta el problema que els temps d'execució són massa elevats. En aquest apartat intentarem emprar el paral·lelisme per tal de reduir aquests temps.

Fent una valoració preliminar, sembla que ens trobem davant un algoritme altament paral·lelitzable, i el primer que haurem de fer és un estudi sistemàtic de les parts de l'aplicació per a veure quines són susceptibles de paral·lelitzar-se i quin seria el benefici obtingut (apartat 5.1).

Quan ja tinguem identificades les parts a paral·lelitzar, estudiarem en la secció 5.2 quina arquitectura paral·lela emprar per tal de portar a terme la sincronització i comunicacions, així com l'emmagatzemament de resultats.

Finalment, en l'apartat 5.3 presentarem el disseny de l'aplicació amb els algoritmes emprats. Donarem algun detall de com s'han perfilat aquests algoritmes i les motivacions que ens han portat a fer-los d'aquesta manera.

5.1 Àrees paral·lelitzables

El primer que caldrà fer és identificar aquelles parts del sistema de calibratge sèrie que són susceptibles de ser paral·lelitzades, és a dir, aquelles tasques que no guarden una relació de dependència temporal entre elles i per tant es poden executar alhora.

Per a realitzar aquesta tasca, no només ens hem de centrar en aquelles parts que programem nosaltres, sinó que hem de tenir consciència també d'aquelles porcions de codi que s'incorporen o són utilitzades per el nostre sistema; ens referim als motors de simulació o les llibreries de resultats.

Després de fer l'estudi, considerem que hi ha 5 punts susceptibles de millora emprant la tecnologia paral·lela:

- **Generació de fitxers**: es tracta de paral·lelitzar la generació del fitxer de l'escenari. Les operacions d'entrada i sortida a disc, acostumen a ser lentes, per tant, la generació de molts fitxers d'escenaris podria ser una tasca molt costosa. Donat que amb només conèixer el codi de l'escenari i

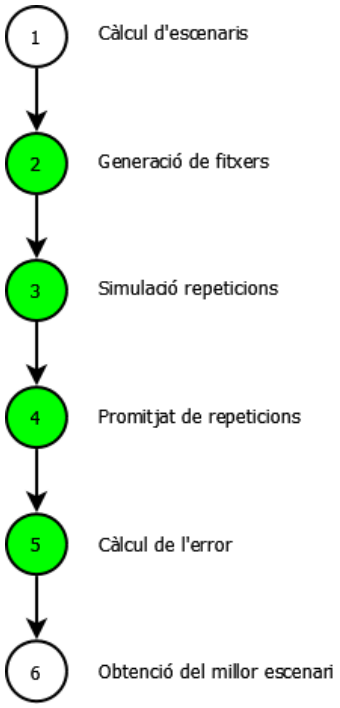

Figura 5.1: Antecedència en les tasques.

poder reconstruir la taula de calibratge es podria generar el fitxer d'aquest escenari, es considera l'opció de delegar aquesta funció al procés que simuli l'escenari.
Per a comprovar si aquesta és una opció vàlida, s'ha realitzat un estudi sobre el cost de generació d'un fitxer a partir de la taula de calibratge (i evidentment els fitxers `.pli` i `.var`) i el cost de construir aquesta taula a partir del fitxer `.cal`. Els resultats han estat els següents:

- Reconstrucció taula calibratge: $0,01$ ms
- Creació de fitxer d'escenari: $20,03$ ms

Segons aquestes dades, si hem de generar 1000 fitxers d'escenaris, fent-ho de forma sèrie tardaríem:

$$0,01 \text{ ms} + 1000 \text{ escenaris} \cdot 20,03 \text{ ms} = 20030,01 \text{ ms}$$

En canvi, si ho realitzem de forma paral·lela, cada procés hauria de crear la taula i escriure el fitxer. D'aquesta forma, si suposem que tenim 1000 nodes lliures, el cost seria:

$$0,01 \text{ ms} + 20,03 \text{ ms} = 20,04 \text{ ms}$$

I si som més realistes i suposem que tenim únicament 20 nodes:

$$50 \cdot (0,01 \text{ ms} + 20,03 \text{ ms}) = 1002 \text{ ms}$$

Clarament representa una millora en el temps de còmput global.

- **Distribució d'escenaris**: donat que els diferents escenaris no tenen cap tipus de dependència entre ells, aquesta opció consisteix en simular els diferents escenaris alhora. De fet, aquesta part del sistema és completament paral·lelitzable, és a dir, si tinguéssim tants nodes disponibles com escenaris a simular, els podríem simular tots alhora.
  Evidentment,s'ha de considerar el cost d'enviar cada escenari al processador que l'hagi de simular i el de preparar els fitxers per cada escenari, però tot i això, aquesta és una tasca altament paral·lelitzable.
  Una possible forma de veure aquesta situació és imaginar una cua d'escenaris a simular, i un *pool* de nodes que simulen escenaris. Conforme aquests nodes estiguin lliures, aniran accedint a la cua per a adquirir un nou escenari a tractar.
  No podem oblidar que cada escenari és simula diverses repeticions, i que després d'aquestes s'obté una mitjana de resultats, i finalment de comparar aquesta mitjana amb els valors reals de referència, s'obté l'error. Aquestes poden ser tasques també a paral·lelitzar, enlloc de realitzar-les de forma sèrie.
  Aquesta proposta ens permetrà paral·lelitzar la tasca 3 de la figura 5.1.

- **Simulació de les repeticions**: per cada escenari realitzarem un nombre determinat de simulacions (normalment entre 30 i 100) i després farem la mitjana dels resultats per tal de salvar les possibles desviacions produïdes per l'atzar intrínsec en els `P-Systems`. Donat que aquestes simulacions són independents les unes de les altres, es pot pensar en executar-les de forma paral·lela.
  Seguint amb el mateix model d'abans, tindríem una cua de repeticions de tots els escenaris, i el que recollirien els processos simuladors seria aquestes repeticions. Acumularien els resultats parcials en una determinada localització, i després un procés s'encarregaria de fer els promitjos de les simulacions i trobar l'error.
  Aquesta proposta és una forma alternativa de paral·lelitzar la tasca 3 de la figura 5.1 respecte la presentada en el punt anterior.

- **Promitjat de les repeticions**: un cop realitzades les simulacions s'ha de fer una mitjana dels resultats parcials per tal d'obtenir els resultats pròpiament dits de cada escenari. Aquesta tasca es troba representada en la figura 5.1 amb el número 4, i també pot ser paralelitzada.
  Entenem, però, que aquesta alternativa només té un sentit veritable si s'ha adoptat la mesura de paralelitzar les repeticions de cada escenari, i no si s'ha optat per fer-ho dels escenaris com una unitat global. Diem això, perquè en cas d'aplicar aquesta proposta, un dels principals problemes és la detecció de quan s'han finalitzat les simulacions de tots els escenaris. Aquesta sincronització serà difícil i segur que tindrà un cost, que repercutirà negativament en el rendiment del sistema.
  Si deleguen les repeticions de cada escenari a un sol procés serà fàcil saber quan s'ha acabat totes les repeticions, ja que s'executen de forma sèrie i controlada. Al finalitzar-les, el mateix node podria executar la tasca de promitjat. Si hem repartit les repeticions, en canvi, haurem de definir un sistema que detecti quan s'han acabat les simulacions de les repeticions d'un mateix escenari i llanci un procés paral·lel que faci les mitjanes només de l'escenari corresponent.

- **Càlcul de l'error**: un cop obtingut els resultats definitius de cada escenari, també es pot paralelitzar el càlcul de l'error que comet cada escenari respecte a la realitat. Aquesta tasca correspon al número 5 de la figura 5.1.
  No es recomana aquesta parelelització com a tasca independent, és a dir, crear un procés per cada escenari que només calculi l'error d'aquest escenari, ni crear un procés que calculi l'error de tots els escenaris, ja que aquesta decisió representaria massa cost de la creació del procés en comparació amb el cost de càlcul de l'error.
  Més aviat recomanem mantenir aquesta tasca dins del procés que simula l'escenari i en calcula la mitja. Aquesta tasca aniria just després de calcular la mitja, així que no caldria cap tipus de sincronització; només seria necessari indicar quan ha acabat de calcular l'error per a indicar que el procés de l'escenari ha finalitzat.

- **Motor de simulació**: una iniciativa molt interessant és aplicar el paralelisme a nivell del motor de simulació. Això significaria evolucionar el model sobre un motor que s'estigui executant entre diversos nodes.
  D'aquesta forma es podria minimitzar el temps en fer les simulacions i per tant el temps total en obtenir les resultats del procés de calibratge.
  L'aplicació d'aquesta tècnica cau fora de l'àmbit d'aquest projecte, però si el lector està interessat en obtenir més informació, pot consultar algunes propostes realitzades per l'equip d'investigació de la Universitat de Sevilla [21].

Per a la realització d'aquest projecte ens hem centrat en l'estratègia de distribuir els escenaris entre els diferents nodes. Aquesta decisió busca un millor balanç entre còmput i comunicacions. Si treballéssim el paral·lelisme a nivell de repetició, necessitaríem realitzar massa comunicacions i sincronitzacions per la poca càrrega que representa la simulació d'una única repetició. Així, es tractaria d'un sistema amb una granularitat massa fina, i no seria eficient.
L'escenari, en canvi, representa una unitat suficientment gran per a justificar una sincronització i una comunicació, però alhora no és massa gran com per ocupar un node durant molt temps.

A més a més, separar-ho en repeticions implicaria definir un mecanisme de sincronització per indicar quan aquestes repeticions han estat simulades, per a poder recollir els resultats. Aquest sistema ens sembla que arribaria a ser força complex, i per tant encara augmentaria més la descompensació entre càrrega i comunicacions.

En definitiva, la nostra aplicació repartirà els escenaris entre els nodes, i cada node simularà totes les repeticions de l'escenari, i en acabar aquestes calcularà la mitjana dels resultats parcials i finalment l'error comés per l'escenari. Quedarà en mans d'un altre procés trobar l'escenari que ha produït un error mínim.

5.2 Arquitectura paral·lela

En aquest apartat prendrem diverses decisions de com estructurar la nostra aplicació paral·lela. Explicarem breument el paradigma Master-Worker triat com a marc general d'aquesta aplicació i després escollirem una forma d'implementar-lo tenint en compte 2 ítems: la sincronització (apartat 5.2.1) i les comunicacions

(apartat 5.2.2). Finalment veurem la solució proposada.

Volem remarcar que la decisió de la solució proposada ha estat fruït de l'anàlisi tant de la sincronització com les comunicacions d'una forma conjunta. En cap cas es pot considerar que són una tria separada, si no que s'ha d'escollir una combinació de les dues per a dissenyar l'arquitectura final. Cap té una prioritat superior a l'altra, i l'ordre en el que es mostren no és vinculant.

Paradigma Master-Worker

El paradigma Master-Worker és una forma de distribuir les tasques en un sistema paral·lel. Es basa en una forma de distribució de les responsabilitats similar a la que es dona en els llocs de treball, amb "encarregatsï "treballadors".
Essencialment defineix 2 rols:

- **Master**: es tracta de l'agent que gestiona el sistema i que s'encarrega de distribuir les tasques amb computació intensa entre els Workers. Normalment és l'element que primer s'executa per tal de planificar què han de fer els diversos treballadors que té a la seva disposició. També és l'encarregat de rebre els resultats de les computacions dels Workers i prendre decisions o mostrar resultats a partir d'aquestes.

- **Worker**: és un "treballador"que serà qui realitzi les tasques paral·leles computacionalment intenses. No prendrà en cap cas decisions estratègiques ni complexes, sinó que delega aquesta responsabilitat al Master.
  Com més independència hi hagi en la tasca a realitzar per cada Worker, més fàcil serà la paral·lelització dels processos i millor resultat s'obtindrà.

Un sistema amb el paradigma ben aplicat disposarà d'un alt nombre de Workers per cada Master, ja que aquest últim no executarà còmput, sinó que tractarà només activitats de gestió.

Donat que el model d'aplicació del calibratge fa que les tasques de cada Worker (simulació d'un escenari) puguin ser independents de la resta, trobem que les comunicacions són bidireccionals entre Master i Worker. Aquesta comunicació bidireccional servirà tant per enviar ordres d'execució com dades per a l'execució i resultats de la mateixa.

5.2.1 Sincronització

Un punt molt important a l'hora de definir l'arquitectura de l'aplicació és la forma en la que els diferents agents del sistema (Master i Workers) duran a terme la seva sincronització. El cas més aparent és com sabrà el Master que els seus Workers han finalitzat l'execució.
Escollir bé aquesta característica afectarà molt significativament en el cost temporal de l'execució, ja que hi ha una gran diferència sobre els recursos emprats i l'entorn necessari entre cada una de les opcions que presentarem.

Hem de tenir present que estem parlant d'una sincronització única, és a dir, que només es produirà una vegada. La idea és notificar al Master quan els Workers han finalitzat la seva execució, per tant evitarem afegir un gran sobrecost amb sincronitzacions molt complexes que no utilitzarem; volem la solució més lleugera possible.

Existeixen moltes alternatives a la bibliografia, però essent realistes amb els recursos que tenim al nostre abast, i quines són factibles d'implementar, n'hem estudiat en profunditat 3:

- **Pas de Missatges**: es tracta d'una aplicació única (probablement realitzada amb `MPI`) que tindrà una part del codi per al Master i una altra part per al Worker. La sincronització es realitzarà a través de missatges, tal i com es pot veure en la figura 5.2.

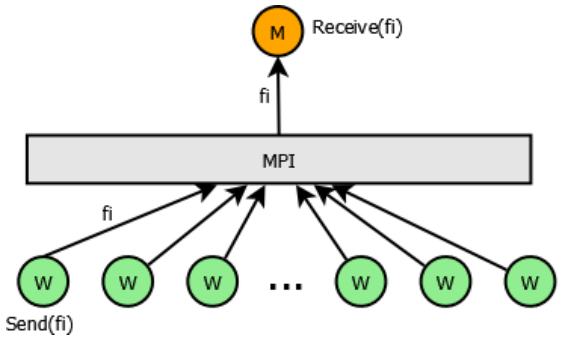

Figura 5.2: Sincronització per pas de missatges.

- ***Standalone***: es tracta de la creació d'una apliació independent que s'executi en els nodes del cluster i que es comuniqui de forma autònoma. Seguint la topología descrita en [17], aquesta sincronització podria ser a través de missatges de xarxa (figura 5.3, dreta) o bé a través de *Remote Procedure Call (*`RPC`*)* (figura 5.3, esquerra).

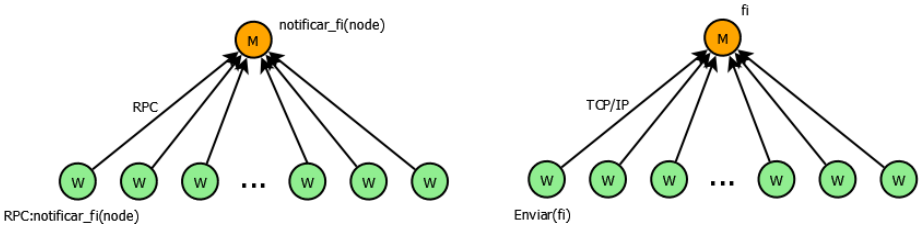

Figura 5.3: Sincronització d'aplicacions independents.

Això implica instal·lar dimonis en la màquina on s'executa el Master i obrir un port a l'espera de rebre peticions.

- **Cues del *Distributed Resource Manager (DRM)***: consisteix en utilitzar el sistema de cues del propi del gestor de recursos del clúster per a sincronitzar la finalització dels Workers. Aquest sistema incorpora una opció que permet encuar un treball i condiciona la seva execució a la finalització d'altres treballs prèviament identificats.
Concretament permet definir aquests treballs a partir del identificador del treball (JobID) o bé a partir del nom del treball (*Job Name*). Aquesta solució no requereix directament una comunicació del Worker amb el

Master, sinó que directament sabrem que el treball que s'ha encuat de forma condicionada només s'executaran quan tots els Workers definits hagin acabat. Podem veure un esquema del seu funcionament en la figura 5.4.

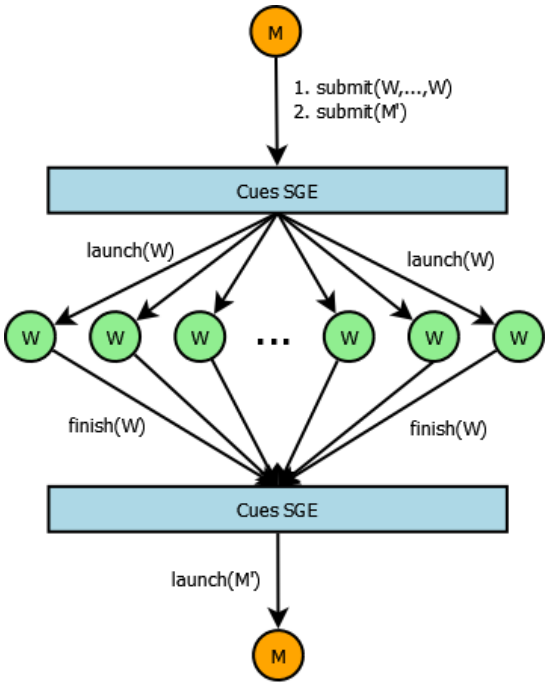

Figura 5.4: Sincronització per cues del SGE.

Sembla fàcil pensar que, en cas d'utilitzar aquesta opció, el Master podria encuar tots els Workers i després encuar una tasca condicionada a la finalització de tots els Workers que l'avisés a ell que han acabat. Això, però, ens faria utilitzar en excés les comunicacions.
Una solució més elegant podria ser que el Master encués tots els Workers i s'encués a si mateix condicionat a la finalització de tots els Workers. Així no hauria de notificar ningú que els Workers han acabat, sinó que al executar-se de nou sabria que això ja ha succeït.

Com hem vist, es tracta de 3 opcions ben diferenciades. La primera és la més utilitzada en la majoría d'aplicacions paral·leles, ja que `MPI` proporciona gran varietat de funcions per comunicar i sincronitzar processos paral·lels executant-se en entorns distribuïts. La segona és més independent, però requereix dissenyar un sistema completament propi amb dimonis que s'han d'aixecar, activació de ports i notificacions, ... Finalment, la tercera és la més lleugera i senzilla de programar i, un cop desplegats els Workers, només requereix llençar una tasca condicionada a la finalització d'aquests.

5.2.2 Comunicacions

El següent punt a contemplar és la comunicació entre el Master i el Worker. Aquesta comunicació ens preocupa en tant que és el moment en que es transme-

tran tant l'escenari a simular pel Worker com els resultats d'aquesta simulació.

Com ja hem dit, es tracta d'una decisió condicionada també a la decisió de la sincronització, ja que si establim un sistema per a sincronitzar els agents a través de missatges, la conseqüència lògica seria utilitzar missatges també per a les comunicacions, i no implementar un altre sistema.

Anem a observar ara les diferents opcions que tenim independentment de la tria que hagem fet en l'apartat anterior. Si una opció representa algun avantatge condicionada a algun tipus de sincronització en concret, ho farem notar:

- **Pas de missatges**: es tractaria de crear una aplicació monolítica amb codi tant pel Master com pel Worker, probablement amb `MPI`. Aprofitaríem l'entorn de pas de missatges per a poder enviar del Master al Worker un missatge amb l'escenari a executar i que el Worker enviés al Master el resultat de l'execució d'aquest escenari. Podem veure un esquema d'aquesta última comunicació a la figura 5.5.
  Val a dir que si s'ha optat per una sincronització per pas de missatges, en el mateix missatge de sincronització es podria incloure el resultat de la simulació. Així, només quedarien dos missatges, un amb l'escenari a executar i l'altre amb el resultat (i que també notificaria la finalització).

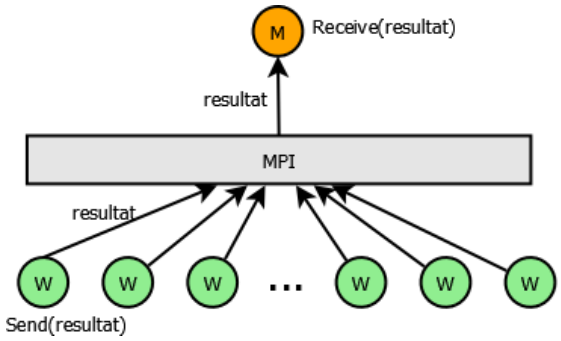

Figura 5.5: Comunicació per pas de missatges.

- ***Remote Procedure Call* (`RPC`)**: consisteix en fer arribar les dades a través de l'execució de codi en màquines remotes. Així, el Master podria tenir una operació `RPC` que fos `enviar_resultat(resultat)` que el Worker executaria i passaria per paràmetre el resultat de la seva simulació. Podem veure aquest esquema en la figura 5.6.
  De la mateixa manera, el Worker podria tenir una operació `RPC` que fos `assignar_escenari(escenari)` per a que el Master li entregués el codi de l'escenari a simular.
  De la mateixa manera que el cas anterior, si hem optat per una sincronització a través de `RPC`, tindria molt sentit utilitzar el mateix mètode en el cas de la comunicació. De fet, també podríem reduir el nombre de operacions `RPC` a 2: una per al llançament i l'altra per la sincronització més el retorn de resultats.

- **Fitxers**: en aquest cas, les comunicacions es realitzarien deixant fitxers degudament codificats en una localització determinada. Així, el Worker

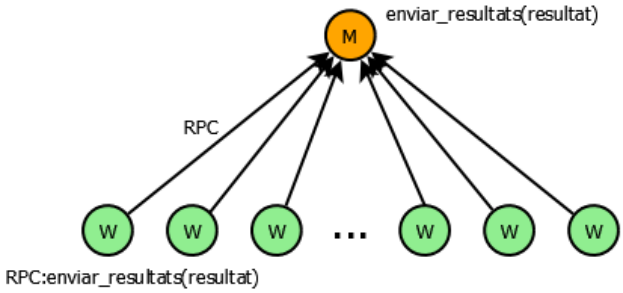

Figura 5.6: Comunicació a través de RPC.

llegira d'aquests fitxers l'escenari que ha d'executar (si prèviament el Master ho ha escrit) i el Master podria recuperar l'error d'un fitxer on hagi escrit el Worker (veure figura 5.7).
Aquesta solució és altament dependent d'una bona sincronització, ja que hem d'estar segurs que els fitxers han estat definitivament escrits si només hi fem una lectura, o bé realitzar una modificació constant per a veure quan han estat efectivament escrits. Aquesta segona opció també podria servir com a sincronització, tot i que ens sembla poc segura.

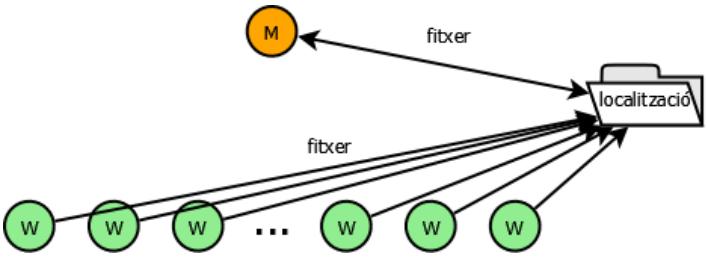

Figura 5.7: Comunicació a través de fitxers.

- **Base de Dades**: l'última opció plantejada és la d'utilitzar una base de dades per tal de que els Workers guardin els resultats de les execucions i el Master, un cop detecti que aquests han acabat, executi una senzilla consulta SQL per tal de trobar l'escenari que ha produït un error mínim. Podem trobar un esquema d'aquest cas a la figura 5.8.
  Aquesta solució no sembla gaire encertada per al repartiment inicial d'escenaris, però sí que es veu com una bona opció per tal de recollir les dades; en cas de voler optar per aquesta, s'hauria de cercar una millor opció per a la distribució d'escenaris com, per exemple, el pas per paràmetre del codi d'escenari al encuar-lo al clúster.

  Pot semblar que la utilització d'una base de dades ha d'afegir un gran cost temporal a la nostra aplicació, però hem trobat diverses implementacions [18] de *lightweight databases* que redueixen molt el temps d'un nombre raonable de consultes i insercions. A més a més, simplifica realment (i accelera de forma notable) la cerca del millor escenari, ja que les bases de

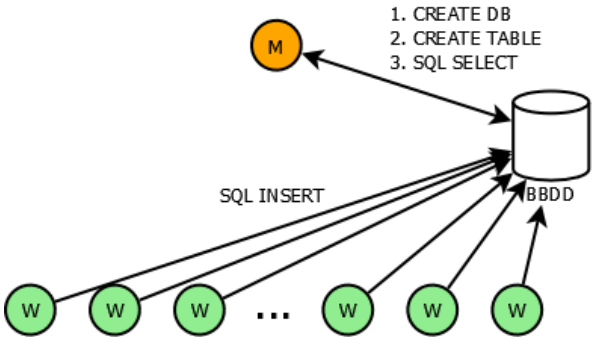

Figura 5.8: Comunicació a través de Base de Dades.

dades tenen diversos índexs optimitzats per a cercar mínims.

5.2.3 Solució proposada

D'entre totes les propostes exposades necessitem escollir una arquitectura fàcil de gestionar que ens permeti assolir una gran eficiència per tal de minimitzar el temps de solució del problema combinatori i augmentar l'escalabilitat del problema.

Degut a la poca quantitat de sincronització que s'ha de dur a terme, i la única situació en la que és realment imprescindible comunicar els resultats, hem optat per ometre les opcions que impliquen `MPI`. Al principi havia sigut una opció prioritària per a nosaltres, fins i tot vam realitzar un estudi sobre les diferents llibreries `MPI` per a `Python` que podem trobar en l'apèndix E.
La majoria d'aplicacions paral·leles han estat desenvolupades amb `MPI`, per això també volíem investigar una nova forma de fer les coses, i no utilitzar el mateix que ja s'ha utilitzat amb anterioritat, sempre mantenint la idea que si l'opció escollida no funciona, es pot canviar l'arquitectura per rectificar aquesta decisió.

La creació de sistemes `RPC` no és una cosa trivial, i requereix conèixer en tot moment les adreces de tots els nodes. Això representa un problema en un clúster on les tasques s'assignen dinàmicament als diferents nodes, i el Master no deixa de ser una tasca, així que sembla impossible conèixer d'avant mà a quin node s'assignarà.
L'altra opció és situar el Master en un servidor independent, amb una adreça pròpia dins la xarxa de comunicacions, i que els Workers es connectessin a aquesta adreça. Això soluciona el problema del retorn dels resultats -tot i que d'una forma un tant complexa- però no resol el problema de la distribució dels escenaris. És clar que podríem passar el codi de l'escenari com a paràmetre al encuar el Worker, així no caldria un servidor `RPC` al Worker.
Tot i aquesta solució, sembla que haver de muntar un servidor independent per tal de rebre els resultats de les execucions és una mesura un tant costosa per a tantes poques dades, ja que a fi de comptes, es tracta de només un número per cada escenari.

Finalment, hem decidit decantar-nos per la sincronització a través de les cues del `DRM` i el retorn dels resultats a través d'una base de dades. En la figura 5.9 mostrem un esquema d'aquesta arquitectura.

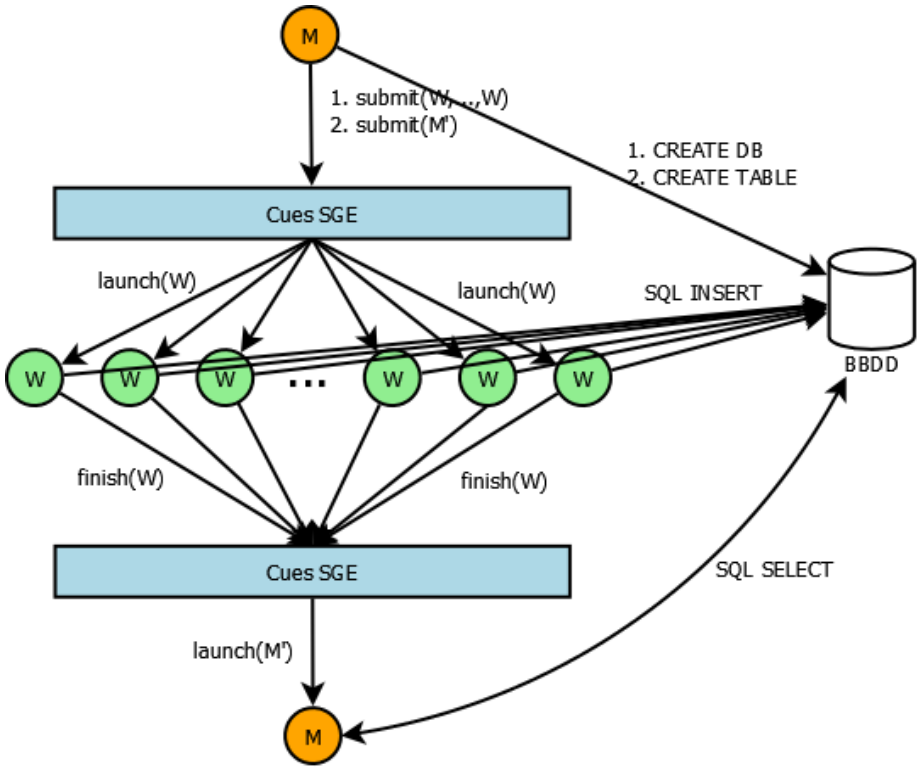

Figura 5.9: Arquitectura de l'Aplicació Paral·lela.

Bàsicament, el que farem és llençar una tasca que serà el Master. Aquesta tasca crearà la base de dades i encuarà tots els Workers passant-los el codi de l'escenari que han de simular per paràmetre. En acabar d'encuar-los tots, s'encuarà a si mateix un altre cop de forma condicionada a que finalitzin tots els Workers.
Els Workers, al seu torn generaran els fitxers de l'escenari i el simularan. Un cop finalitzada l'execució de totes les simulacions, faran la mitjana dels resultats i calcularan l'error que ha comés l'escenari i introduiran aquest error en la base de dades. En aquesta solució hem optat per la creació dels fitxers, el càlcul de la mitjana i la determinació de l'error en el mateix Worker, ja que així ens estalviem moltes sincronitzacions innecessàries per tant poca càrrega computacional.
Quan hagin finalitzat tots els Workers, el sistema de cues DRM llençarà el Master que s'ha encuat condicionat a la finalització dels Workers llençats, moment en el que Master ja tindrà accés a la base de dades amb la seguretat que tots els Workers han acabat la seva execució i hi han inserit els seus resultats. Aleshores, el Master farà una consulta `SQL` per a trobar l'escenari que ha produït l'error mínim i el retornarà.

A continuació explicarem el disseny que hem realitzat per a la base de dades.

Base de Dades

Un bon disseny de la base de dades pot representar un canvi substancial en el cost i la utilitat d'ús d'aquesta. Així, reflexionar detingudament sobre quines són les necessitats de la base de dades i com implementar-la paga la pena per tal d'obtenir uns rendiments superiors.

El principal motiu per el que volem disposar d'una base de dades és per a poder **retornar** de forma desatesa **els resultats** de les execucions dels Workers. Més enllà d'això, i aprofitant aquest fet, també la podem utilitzar per a **estalviar** en l'execució de còmput.
Al llarg del desenvolupament d'un model, aquest es pot veure modificat lleugerament per a millorar-ne els resultats. Un canvi molt comú pot ser, per exemple, canviar els rangs de calibratge de les variables. Si disposem d'una base de dades amb els escenaris de cada model executats amb anterioritat, podríem comprovar que cada escenari no hagi estat simulat amb anterioritat.

Un altre motiu que ens pot interessar és de cara a **estudis estadístics** dels resultats de les simulacions dels escenaris. Per exemple, podem voler representar l'error produït per els escenaris en tot l'espai de cerca dels mateixos, o bé només mirar com un paràmetre en concret afecta en els resultats de la simulació. Per a poder fer aquests estudis, necessitarem els errors de tots els escenaris. Per tot això es fa imprescindible triar un bon identificador per als escenaris.

És important triar un bon tipus de base de dades. Degut a la baixa exigència en inserció de dades i lectura de les mateixes, ens plantegem l'ús d'una *lightweight database*. Aquestes bases de dades estan basades en fitxers i un motor incorporat en el sistema de còmput. Bàsicament, es pot crear una base de dades directament en un fitxer, a més a més, aquest es pot moure entre sistemes i ser llegit en el nou sistema.
Concretament nosaltres hem optat per la implementació de `SQLite`, molt estesa en servidors amb pocs requeriments de dades o bé en clústers que busquen independència de les dades respecte una arquitectura de servidors `SQL` concreta.

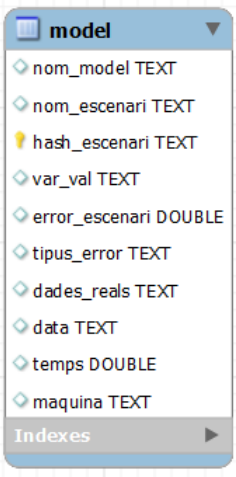

Figura 5.10: Disseny de la base de dades.

En la figura 5.10 podem veure els camps que formaran cada taula. Definirem

una taula per cada model (en el de la imatge "newtoctubre"), de la manera que tindrem tots els escenaris d'un model en la mateixa taula.

De les converses amb els futurs usuaris d'aquesta aplicació, hem acordat que si es realitzat un canvi en el model, s'haurà de canviar el nom d'aquest. Aquest és el procediment que han estat seguint fins ara, i això ens estalvia haver de definir una forma d'obtenir identificadors únics per cada model.
Essencialment, s'han establert dos identificadors:

- **Model**: l'identificador del model serà el seu nom. Si es canvia lleugerament aquest model (regles, variables, etc.), s'haurà d'optar per un nom diferent. Aquest nom serà definit per l'usuari de l'aplicació.
- **Escenari**: per a identificar l'escenari emprarem una eina de *hash*. Concretament, i considerant que ja hem identificat el model al que pertany, farem el *hash* (amb la funció *SHA256*) de totes les variables de l'escenari amb el seu valor, de la forma:

  $$sha256(\text{“}\mathtt{m\{0\}} = 0.08, \mathtt{m\{1\}} = 0.07, \mathtt{m\{2\}} = 0.07, \ldots, \mathtt{p\{1\}} = 0.55, \ldots \text{“})$$

  Així podrem identificar fàcilment quins *hashs* coincideixen amb els que ja es troben emmagatzemats en la base de dades i no repetir el còmput. Calcula la funció *hash* d'una cadena de caràcters és una funció de baix cost i d'evolució lineal.

Sempre i que es mantingui aquesta norma, el sistema pot estalviar la repetició en el tractament dels escenaris, estalviant-se així un nombre substancial de simulacions i millorant molt el rendiment de l'algoritme de calibratge.

L'elecció dels camps de cada taula ha estat també el fruit de la reflexió conjunta entre els desenvolupadors d'aquesta aplicació i les necessitats indicades per els futurs usuaris de la mateixa (els usuaris ecòlegs). D'aquesta forma s'ha arribat a un consens sobre el nombre i la naturalesa de dades a guardar. A continuació trobem una descripció de tots els camps:

- Nom del model (`nom_model`): es tracta de guardar el nom del model que ha definit l'usuari. Aquest nom serà l'identificador únic del model dins del nostre sistema.
- Codi de l'escenari (`nom_escenari`): és el codi de l'escenari, de la forma `(0,1,3,...,1,2)`. Servirà per a conèixer els valors de les variables, tot i que si es canvia el rang de calibratge d'alguna, ja no ens servirà. Per això, no el considerarem l'identificador únic de l'escenari.
- *Hash* de l'escenari (`hash_escenari`): aquest és l'identificador únic de l'escenari dins d'un model. Es tractarà, per tant, del **camp clau** de la taula. Dos escenaris amb les mateixes variables a calibrar amb els mateixos valors (encara que siguin de models diferents) tindran el mateix *hash*. Per això, només el podrem fer servir per a identificar un escenari dins d'un model.
- Valors de les variables (`var_val`): serà la cadena de caràcters amb les variables a calibrar i els seus valors per aquest escenari. No serà igual a la cadena que emprarem per a calcular el *hash* de l'escenari, ja que aquesta només conté les variables a calibrar i els seus valors. Ho emmagatzemarem per a poder trobar els valors de cada variable en aquest escenari.

- Error de l'escenari (`error_escenari`): es tracta de l'error de l'escenari reportat per el Worker.
- Tipus d'error (`tipus_error`): tot i que en aquesta memòria només hem parlat de la distància euclidiana, hi ha altre formes de calcular els errors dels escenaris. Aquí indicarem amb una cadena de caràcters quina ha sigut aquesta forma.
- Nom de les dades reals (`dades_reals`): es tractarà del nom del fitxer de les dades reals per a aquest model. De la mateixa forma que el nom del model, si canvia aquest fitxer (cosa realment improbable) haurà de canviar el seu nom.
- Data (`data`): data d'execució de la simulació.
- Temps d'execució (`temps`): temps que ha tardat el Worker en realitzar la simulació de l'escenari i en calcular l'error.
- Descripció de la màquina (`maquina`): descripció del clúster on s'està executant l'aplicació.

Un cop hem vist l'arquitectura de l'aplicació, és el moment de veure com funcionarà la mateixa i descriure els diferents algoritmes que la descriuen.

5.3 Disseny de l'aplicació

El correcte disseny de l'aplicació ens permetrà desenvolupar-la molt més fàcilment. El que farem és, basant-nos en l'arquitectura de l'aplicació que hem definit en l'apartat 5.2.3, dissenyar una aplicació que realitzi les tasques requerides.

Recordem que aquesta aplicació es basarà en el paradigma Master-Worer i que per tant tindrà dos agents. Cada agent disposarà d'unes responsabilitats i desenvoluparà unes tasques. Podem veure una representació del flux general d'operacions en la figura 5.11.

Cronològicament parlant, el funcionament és el següent:

1. Es llença el Master.
2. El Master crea la base de dades.
3. El Master calcula el nombre d'escenaris i llença un Worker per cada un. Li passarà al Worker el codi de l'escenari que ha d'executar per paràmetre.
4. El Master s'encuarà a si mateix condicionant la seva execució a que tots els Workers hagin acabat la seva execució.
5. Els Workers simulen els escenaris i guarden els resultats a la base de dades.
6. Els Workers finalitzen l'execució i el DRM llença un altre cop el Master.
7. El Master consulta a la base de dades l'escenari amb error mínim i l'anuncia com a millor.

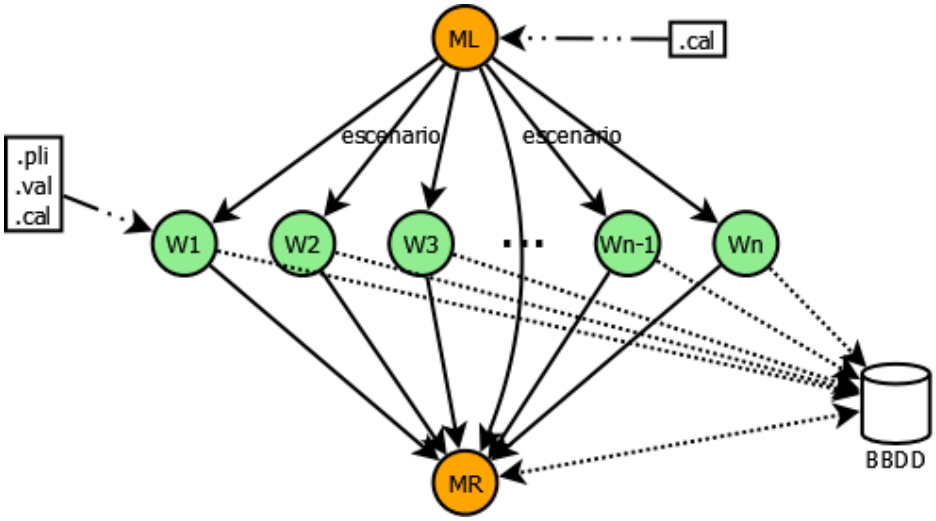

Figura 5.11: Disseny de l'aplicació paral·lela.

Per a tenir una idea més clara del funcionament de les diferents tasques encuades en el sistema, hem preparat un petit esquema que ens indica quins *scripts* (`master.sh` i `worker.sh`) de enviament de tasca tenim i com es relacionen amb les dues aplicacions `Python` que hem preaparat: la master (`master.py`) i la worker (`worker.py`). Podem veure aquestes relacions i també amb la base de dades en la figura 5.12.

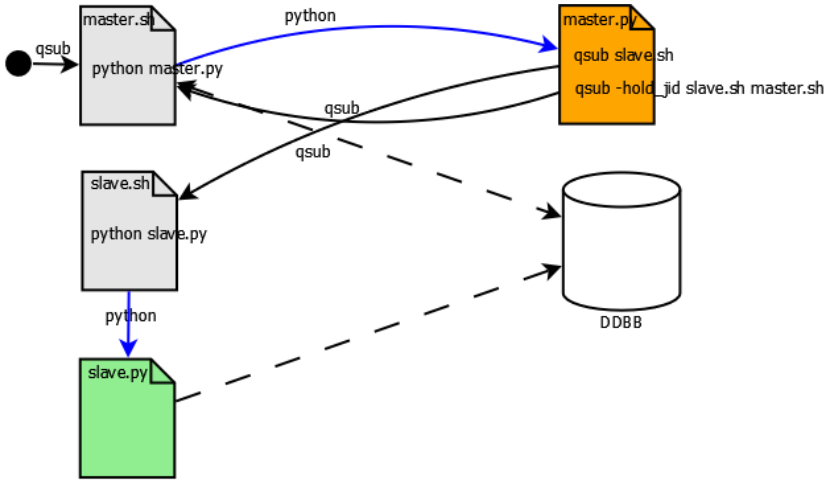

Figura 5.12: Esquema de *scripts* i fitxers de codi.

A continuació analitzarem amb detall el paper que juga cada un dels agents del sistema. Veurem l'algoritme que segueix el Master i també quines són les operacions seran realitzades pels Workers. Podem trobar la implementació d'a-

questa aplicació en l'apèndix C.

5.3.1 Master

El Master realitza dues accions molt diferenciades. La primera és llençar tots els Workers per a que realitzin la simulació dels escenaris. La segona, en canvi, tracta de trobar el millor escenari en la base de dades.

En la primera ocasió que llancem el Master s'haurà de fer la distribució dels escenaris, en canvi, la segona serà quan recollirem els resultats. Això ho podríem fer amb dos tasques diferents, o bé amb una única tasca que diferenciï si es tracta de la primera ocasió que s'ha llençat o la segona per mitjà d'un paràmetre. Per a reduir el nombre de tasques, hem optat per la aquesta última versió, tal i com podem veure en l'algoritme 4.

Algorithm 4 Algoritme Master global.

Require: fitxer: `.cal`

```
1: if rellançat then
2:     recollir_resultats();
3:     return
4: end if
5: distribucio_escenaris();
6: rellançar_master(rellançat);
```

Ara passarem a analitzar els detalls de cada un dels algoritmes: el de recollir resultats (algoritme 6) i el de distribuir escenaris (algoritme 5).

En l'algoritme 5, podem veure els passos a seguir per a iniciar la simulació de tots els escenaris. El primer que farà és inicialitzar la configuració i llegir el fitxer `.cal` (línies 1 i 2). Després crearà la base de dades (línia 3) en una localització determinada. Aquesta localització haurà de ser coneguda també pel Worker i hi hauran de tenir accés. El següent és reconstruir la taula de calibratge (línia 4) i inicialitzar els iteradors (línia 5).
Aleshores calcularem el nombre d'escenaris (línia 6) i iniciarem el bucle per a recorrer-los tots (línies 7-11). Per cada escenari,

- Generarem el identificador de l'escenari (línia 8).
- Llençarem un Worker passant-li aquest identificador (línia 9).
- Calcularem el següent escenari.

Acabarem l'execució d'aquest algoritme rellançant el codi del Master condicionat a que s'hagi finalitzat l'execució de tots els seus Workers.

En l'algoritme 6 podem veure l'algoritme de la segona execució del Master, es a dir, el de la cerca del millor escenari. El primer que farà és connectar-se a la base de dades (línia 1) que es trobara a la *localitzacio* indicada anteriorment. A continuació farà una consulta `SQL` a aquesta base de dades per a trobar el mínim error, i l'escenari que l'ha causat (línia 2). Finalment, en la línia 3, retornarem aquest $escenari_{min}$ i $error_{min}$.

Algorithm 5 Distribuir escenaris - Master

Require: fitxer `.cal`.

```
1: inicialitzar_configuracio();
2: cal := llegir_fitxer_cal(ruta_cal);
3: crear_base_dades(localitzacio);
4: taula := reconstruir_taula_calibratge(cal);
5: it := inicialitzar_iteradors(taula);
6: nombre_escenaris := calcular_nombre_escenaris(it);
7: for (i := 0; i < nombre_escenaris; i++) do
8:    id_escenari := generar_id_escenari(it);
9:    llençar_simulacio(id_escenari, repeticions);
10:   calcular_seguent_escenari(it);
11: end for
12: rellançar_master();
```

Algorithm 6 Recollir escenaris - Master

```
1: connectar_base_datos(localitzacio);
2: escenari_min, error_min := consultar_millor_escenari();
3: return escenari_min, error_min
```

5.3.2 Worker

El Worker només realitza una funció: simular l'escenari. Aquesta funció, però, implica un bon nombre d'operacions, entre les que trobem: generar els fitxers de l'escenari, calcular la mitjana dels resultats i calcular l'error.

En l'algoritme 7, trobarem les operacions que duu a terme el Worker. El primer que cal comentar són els requeriments: necessitarà la configuració de les localitzacions dels fitxers, el nombre de repeticions, ... També li farà falta capturar el paràmetre *codi_escenari* que li passarà el Master i els fitxers `.pli`, `.var`, `.cal`, `.esp` i `.avg` per a poder realitzar les diferents tasques.

Algorithm 7 Algoritme Worker.

Require: Configuració, paràmetre *codi_escenari* i fitxers `.pli`, `.var`, `.cal`, `.esp` i `.avg`.

```
1: inicialitzar_configuracio();
2: llegir_fitxers();
3: connectar_base_dades(localitzacio);
4: taula := reconstruir_taula_calibratge();
5: it := inicialitzar_iteradors(codi_escenari);
6: fitxer_escenari := generar_fitxer_escenari(it);
7: for i in N do
8:    resultats[i] := simular(fitxer);
9: end for
10: resultat := promitjar(resultats);
11: error := distancia_euclidiana(resultat,real);
12: afegir_resultats_base_dades(codi_escenari, error,...);
13: esborrar_fitxers_escenari(escenari);
```

Un cop vistos els requeriments d'entrada, podem passar a analitzar el curs de l'execució. Començarà per inicialitzar la configuració i llegir els fitxers (línies 1 i 2). Després es connectarà a la base de dades per a poder guardar-hi el resultat més endavant (línia 3).
Aleshores reconstruirà la taula de calibratge (línia 4) per a poder saber quines són les variables a calibrar i trobar el seu valor a través del *codi_escenari* que ha rebut del Master (línia 5). Un cop tenim ja els valors de les variables definits, podrà generar el fitxer de l'escenari (línia 6) a partir dels iteradors, la taula de calibratge i els fitxers `.pli` i `.var`.

Ara ja està tot preparat per a realitzar la simulació de l'escenari. Simularem tantes vegades l'escenari con ens ho indiqui la configuració (línies de 7 a 10). Acabada la simulació, farem el promitg dels resultats (línia 10) i ja podrem calcular l'error, en aquest cas a través de la distància euclidiana (línia 11).

Obtingut ja l'error, el guardarem a la base de dades (línia 12) i esborrarem els fitxers generats en el procés (línia 13). Arribats a aquest punt, la tasca finalitzarà i ho notificarà (de forma automàtica i transparent per a nosaltres) al `DRM`. Aquest, quan hagi observat que tots els Workers han finalitzat, despertarà la tasca Master -que havia estat encuada en espera de que tots els seus Workers finalitzin- la qual determinarà l'escenari que proporciona els millors resultats.

Podem trobar una implementació realitzada amb `Python` d'aquesta aplicació paral·lela per a la realització del calibratge automàtic de models en l'apèndix C.

Capítol 6

Experimentació i Resultats

En els capítols anteriors hem definit com serà el nostre sistema de calibratge, tant en la seva versió sèrie inicial com en la versió paral·lela més avançada. Ara és el moment d'escollir un ecosistema real i provar aquest sistema, per veure quant eficient és, quins són els factors millorables, etc.

El primer que farem és descriure a grans trets l'ecosistema triat per a les proves. Després analitzarem l'entorn d'execució de les proves (apartat 6.2) i quines seran aquestes proves a realitzar (apartat 6.3). Finalment analitzarem els resultats de rendiment obtinguts amb cada sistema.

6.1 Ecosistema de referència

L'objectiu d'aquest model és representar l'ecosistema del tritó pirinenc (*Calotriton asper*) que és una espècie endèmica que habita els Pirineus i els Pre-Pirineus. El model es centrarà en el torrent del Vall del Pi que forma part d'un conjunt de 38 torrents, 16 dels quals estan ocupats per el tritó. Proper a aquest torrent, trobarem 8 altres que també influeixen en la seva dinàmica i que, per tant, hauran de ser inclosos en el model d'alguna forma.

El nombre de tritons que habiten un torrent dependrà d'un conjunt considerable de factors biòtics i abiòtics alhora: variabilitat del clima, presència de peixos, antecedents de l'espècie, disponibilitat d'aliments, etc. Cada un d'aquests factors es poden estudiar de forma separada, però necessitem un model que pugui treballar amb tots, ja que tots aquests factors estan relacionats.

El model proposat per a l'estudi del tritó està format per un únic entorn en el que hi ha 9 membranes, una per cada torrent (una pel torrent en estudi i 8 pels torrents que l'afecten). L'estructura d'aquesta membrana es pot representar així:

$$\mu = [[[\,]_{11}]_1[[\,]_{21}]_2[[\,]_{31}]_3[[\,]_{41}]_4[[\,]_{51}]_5[[\,]_{61}]_6[[\,]_{71}]_7[[\,]_{81}]_8[[\,]_{91}]_9]$$

Donat que alguns paràmetres biològics depenen de les característiques físiques del torrent (com ara el pendent, la profunditat, etc.), les regles d'evolució no seran exactament iguals per a tots els entorns físics, és a dir, per a totes les membranes.

A continuació descriurem els 7 mòduls que formen el bucle del model. El pas d'un any de simulació s'entendrà com la execució completa d'una volta del bucle, és a dir, l'execució completa de tots els mòduls:

- Mòdul 1: en el primer mòdul es generen els objectes que guarden informació de l'existència (o la seva absència) d'inundacions. I, en el cas d'inundació, el moment en el que es produeix. De forma simultània, si hi ha hagut una introducció d'animals en els torrents, s'afegeix.
- Mòdul 2: es tracta del mòdul que gestiona les inundacions en primavera, per tant, només s'executa quan realment hi ha inundacions en aquesta època de l'any.
- Mòdul 3: el tercer mòdul és l'encarregat de les regles de reproducció. El model indica que tots els individus en edat fèrtil es poden reproduir de forma aleatòria amb una probabilitat p observada experimentalment.
- Mòdul 4: si en el torrent hi trobem truites de riu, aquestes depredaran part dels ous dipositats per les femelles, així com alguns tritons joves. El quart mòdul s'encarrega de comptabilitzar aquestes depredacions.
- Mòdul 5: també hi ha mortaldat causada de forma natural. Aquests tipus de baixes es recullen en el mòdul 5.
- Mòdul 6: en el cas de patir una inundació a la tardor, aquest mòdul s'executarà. S'intercanvien dades amb l'entorn, però no s'evoluciona cap objecte. En canvi, aquells que han mort a causa de la inundació són dissolts.
- Mòdul 7: finalment, l'últim model s'encarrega de les migracions entre torrents d'aquells individus en edat terrestre. Es tracta d'un intercanvi d'objectes a nivell d'entorn amb una probabilitat determinada per cada torrent.

Per a validar el model, s'han utilitzat dades des de l'any 1982 al 2002 [19]. S'ha calibrat el model sense l'ajuda del sistema automàtic de calibratge, amb uns resultats força bons [8]. Concretament, estem parlant d'un ajust:

$$R = 0.781, R^2 = 0.61, p < 0.0001$$

On podem considerar que la diferència es deu a l'atzar.
Esperem que l'aplicació del nostre sistema permeti trobar uns resultats inclús millors.

El fitxer de calibratge definit per aquest model (veure D.3) defineix 5 paràmetres a calibrar, amb un total de 768 escenaris possibles. Es tracta d'un model de dimensions relativament petites. Tot i que sense el sistema de calibratge automàtic, els usuaris calibren un nombre més reduït de paràmetres, el fet de poder-ho fer de forma automàtica propicia que el nombre de paràmetres i el seu rang augmenti.

6.2 Entorn d'execució

És força important analitzar l'entorn en que s'han realitzat les proves, ja que d'un sistema de computació a un altre hi poden haver unes grans diferències. Intentarem que l'equip de proves de la versió sèrie sigui el més similar possible a les característiques dels nodes de l'equip de la simulació paral·lela.

L'aplicació **sèrie** s'ha executat sobre un equip amb Intel Core 2 Quad a 2,4GHz i amb 4GB de memòria RAM. Els resultats s'han emmagatzemat en un disc dur en xarxa connectat amb un enllaç de 1Gbps amb un únic salt.

En les proves de l'**aplicació paral·lela** s'ha utilitzat el clúster *Maracas* disponible a la Universitat de Lleida. Es tracta d'un sistema amb 24 nodes homogenis. Cada node té les mateixes característiques que l'equip que hem emprat per la prova sèrie (Intel Core 2 Quad a 2,4GHz i 4GB de RAM). Les connexions entre els nodes estan realitats a través d'una xarxa a 1Gbps. En aquest cas, també hem emmagatzemat els resultats en el mateix disc dur en xarxa, amb el mateix enllaç.

6.3 Proves realitzades

Per tal de comprovar el rendiment del nostre sistema, hem dut a terme diverses proves sobre l'entorn que hem presentat en l'apartat anterior. Aquestes proves aniran dirigides a comparar els resultats de temps de l'aplicació sèrie amb els resultats de temps de l'aplicació paral·lela.

El primer que farem, doncs, és realitzar el calibratge automàtic del model presentat anteriorment (el del tritó pirinenc) en el **sistema sèrie**. Realitzarem aquest calibratge diversos cops i farem la mitjana dels temps resultants, per tal d'obtenir un valor fiable.

Un cop tinguem el temps sèrie, el que haurem de fer és simular el mateix escenari en el **clúster** *Maracas*. Sobre aquest equip, realitzarem calibratges utilitzant diversos nombres de processadors, per tal de veure què tal escala el problema, i com canvia l'eficiència de l'aplicació paral·lela al llarg dels processadors. Intentarem extrapolar els resultats obtinguts a clústers de majors dimensions.

Així, començarem realitzant un calibratge amb els **24 nodes** disponibles al clúster. Aquesta hauria de ser l'execució més ràpida. Continuarem utilitzant només la meitat de nodes (**12**) i seguirem amb **6** i finalment amb **3**. També realitzarem un calibratge amb només **1 node**, però emprant el codi de l'aplicació paral·lela, per tal de veure quant de temps s'afegeix a l'execució a causa del sobre-cost de les comunicacions paral·leles.

Donada l'arquitectura del nostre sistema de calibratge, aquest aprofita sempre tots els recursos disponibles del clúster. Per això, realitzar una execució que només utilitzi una part dels nodes requerirà fer que només el nombre exacte de nodes de la prova estiguin lliures.
Per a fer-ho, llençarem uns processos *dummies* que l'únic que faran és ocupar un node durant un temps infinit. Així la nostra aplicació només disposarà dels nodes que no estiguin corrent aquests processos *dummies*.
Per exemple, per la simulació amb 12 nodes, llençarem 12 processos *dummies* i després llençarem l'aplicació, que només trobarà 12 nodes lliures.

6.4 Resultats

En la taula 6.1 podem observar els resultats obtinguts de les proves descrites en la secció 6.2.

Com podem veure, hem obtingut el temps d'execució de l'aplicació sèrie i de l'aplicació paral·lela en les versions amb 1, 3, 6, 12 i 24 nodes. Per les diferents execucions de la versió paral·lela hem calculat l'acceleració (també conegut com *speedup*) i l'eficiència respecte l'execució de la versió paral·lela amb només 1 node; per això tant l'acceleració com l'eficiència en aquest últim cas val 1.

Concretament l'**acceleració** ens permet conèixer l'augment de velocitat (o, inversament, la disminució del temps d'execució) de l'aplicació a mesura que augmenta el nombre de recursos. L'equació que ens permet calcular aquesta acceleració és la següent:

$$Speedup = \frac{T_{sèrie}}{T_{paral·lel}}$$

És a dir, quantes vegades és superior el temps sèrie al temps paral·lel. Podem imaginar-nos que existeix sempre un límit superior d'aquesta acceleració, i aquest és el nombre de nodes emprats.
Per exemple, una aplicació paral·lela que fa un ús excel·lent de 4 nodes, no pot ser mai més ràpida que 4 vegades la mateixa aplicació amb un sol node.

L'**eficiència**, en canvi, ens permet copsar com de bona és una acceleració, comparada amb l'acceleració ideal que es podria obtenir. En aquest sentit, l'equació és:

$$Eficiència = \frac{Speedup}{Nodes}$$

On l'*speedup* és l'acceleració calculada anteriorment, i $Nodes$ és el nombre de nodes amb els que s'ha executat l'aplicació.
Com ja hem vist, el valor màxim del *speedup* és el nombre de nodes, per tant, el valor màxim de l'eficiència serà 1. Com més proper a aquest valor sigui, millor estarem utilitzant els nodes del sistema i més eficient serà la nostra aplicació.

Taula 6.1: Taula de resultats.

Proc.	Temps (s)	Acceleració	Eficiència
Sèrie	123240	-	-
1	126975	1	1
3	42420	2,993	0,9977
6	21300	5,961	0,9935
12	10620	11,956	0,9963
24	5460	23,255	0,9689

La primera dada de la que volem fer menció és la proximitat en el temps d'execució entre la versió sèrie, i la versió paral·lela. Efectivament,

$$|123240 - 126975| = 3735 segons$$

Generalment, aquest seria el temps perdut en comunicacions i sincronització. En el nostre cas, però, no podem afirmar que tot aquest temps sigui degut a les comunicacions i la sincronització. Això és degut a el fet que el DRM revisa les cues en busca de nous treballs a executar cada un cert període de temps (10 segons en el nostre entorn d'execució). És a dir, en el pitjor dels casos, un

treball podria estar esperant a la cua 10 segons. Aquest temps s'afegix al temps perdut per les comunicacions i la sincronització, fent que sigui força complicat saber quina fracció del total correspon a cada part.
Com podem comprovar, això no ha passat en els 768 treballs enviats, sinó la diferència encara seria major ($768 * 10 = 7680 segons$), però és molt provable que hagi passat a un bon nombre de treballs.
També hem considerat que l'accés a la base de dades i l'obtenció de l'error mínim (una consulta `SQL`) no representa un gran cost per al sistema.

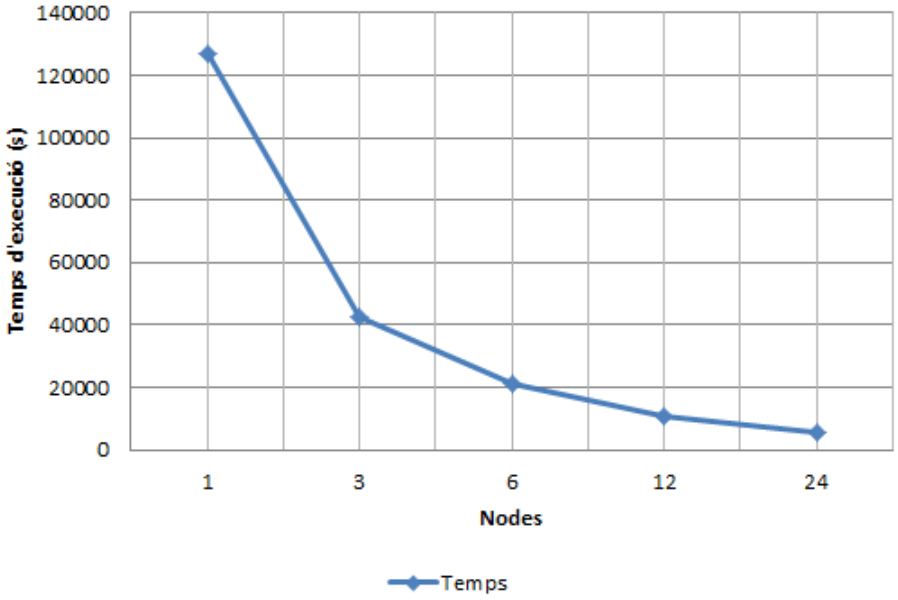

Figura 6.1: Gràfica del temps d'execució.

El següent que cal analitzar és l'evolució del temps d'execució al llarg de les proves amb diferents nodes. En la figura 6.1 observem com, al passar d'un a 3 nodes, el temps d'execució cau pràcticament a un terç. Al passar de 3 a 6, en canvi, disminueix pràcticament a la meitat, i així continua mentre anem doblant el nombre de nodes disponibles per a la nostra aplicació. Podem deduir, doncs, que la disminució de temps és directament proporcional al nombre de nodes disponible i, de fet, força proper a aquest valor.

Per apreciar millor aquest augment de velocitat en la figura 6.2 traçarem el gràfic de l'acceleració. Com podem veure, l'acceleració és pràcticament lineal en funció dels recursos. A més a més, tot i que cap al final apareix una petita disminució, sembla que augmenta exactament igual en cada nova experiència.

Per tal de veure amb més exactitud si l'acceleració és la mateixa que el nombre de nodes, analitzarem la gràfica de l'eficiència, que mesura exactament això; com de proper és l'acceleració al nombre de nodes (veure figura 6.3).

Com podem veure, l'eficiència és pràcticament d'1 en la major part dels casos. La gràfica 6.3 s'ha escalat entre el $0,9$ i el 1 per a veure-ho amb més precisió. Podem observar com entre els nodes 3 i 12 es manté per sobre del

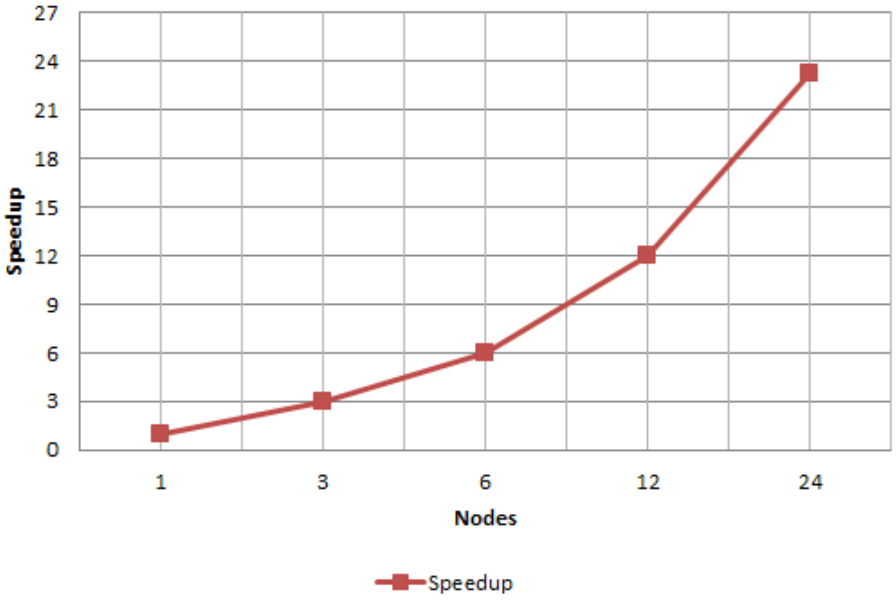

Figura 6.2: Gràfica de l'acceleració o *speedup*.

0, 99, això vol dir que pràcticament estem tractant amb una acceleració ideal.

En el cas dels 24 nodes sembla que empitjora una mica però, tot i això, es troba per sobre del 96% d'eficiència.
Una possible explicació a la lleugera disminució de l'eficiència pot ser que en tots els casos disposem d'uns costos mínims fixes: l'execució de dues vegades del Master (una per llançar els escenaris, i l'altra per trobar-ne el millor). En tots els casos aquests costos es distribueixen entre totes les rondes d'escenaris que s'executen alhora al clúster, però el que succeeix és que conforme augmentem el nombre de nodes, hi ha menys rondes entre les que distribuir aquests costos mínims. Així, aquest cost mínim fa que l'eficiència baixi lleugerament conforme anem augmentant el nombre de nodes; bàsicament perquè augmentem el temps global d'execució.

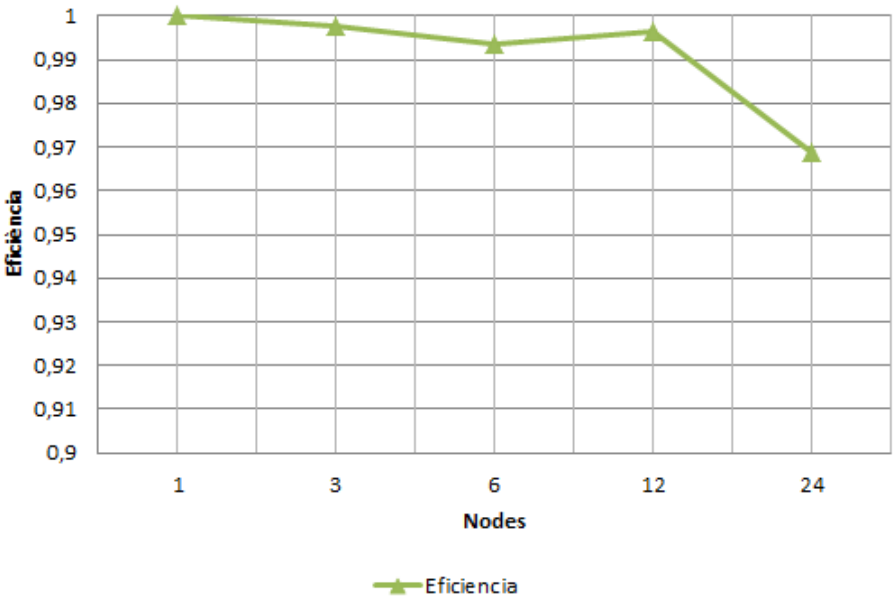

Figura 6.3: Gràfica de l'eficiència.

Capítol 7

Conclusions i Treball Futur

La realització d'aquest treball es va començar amb l'estudi dels `P-Systems`. Hem pogut observar com s'utilitzen avui en dia per a la simulació d'ecosistemes complexos, i com la seva aproximació estocàstica permet aconseguir uns resultats força bons.

Fruit de l'estudi d'aquests models, i del procediment que els ecòlegs segueixen per tal d'implementar-los, hem pogut constatar la necessitat d'un sistema que els permeti el calibratge d'aquests models per tal d'obtenir els millors resultats possibles de les seves simulacions.

Estudiant aquest problema, hem pogut copsar com aquest és un problema amb unes necessitats de còmput molt elevades, ja que es tracta d'un problema que inclou una explosió combinatòria en el nombre d'escenaris a estudiar.

Hem atacat el problema desenvolupant una aplicació sèrie que fes un recorregut exhaustiu per tots els escenaris possibles i trobés el que millor simula l'ecosistema real, però aquesta solució presenta una forta mancança. Donat que el nombre d'escenaris creix de forma escalar amb el nombre de paràmetres a calibrar, el temps que tarda aquesta aplicació sèrie és massa elevat en ecosistemes complexos. Tot i que per a models petits amb poques necessitats de calibratge és vàlida, es fa necessari dissenyar una aplicació que acurti el temps d'execució.

Per a solucionar aquest problema, hem dissenyat i implementat una aplicació paral·lela que utilitzarà tants nodes com tingui disponibles per a realitzar la cerca molt més ràpid. La idea radere d'aquesta aplicació ha estat repartir el nombre d'escenaris a simular entre tots els nodes, per tal de reduir el temps d'obtenció de dades.
La implementació d'aquesta aplicació s'ha pensat de tal forma que aprofiti al màxim els recursos del clúster, i que no es carregui amb complements innecessaris o infrautilitzats. En aquest sentit, hem optat per la implementació de la sincronització de l'aplicació mitjançant les cues del `DRM`, i utilitzar l'opció de cua condicionada. Les comunicacions es faran a través d'una base de dades.

A partir de les proves realitzades, hem obtingut els resultats de temps de l'execució. L'aplicació sèrie tardava 123240 segons en realitzar un calibratge, mentre que l'aplicació paral·lela amb 24 nodes tarda 5460 segons en realitzar la mateixa operació. Hem estalviat 117780 segons, o el que és el mateix, hem reduït 22 vegades el temps d'execució del sistema. Amb l'aplicació paral·lela tardem un $95,57\%$ menys de temps.
En l'estudi sobre l'escalabilitat i l'eficiència del sistema paral·lel al llarg de

diferent nombre de nodes, hem pogut observar com l'escalabilitat es manté força constant, amb acceleracions sempre properes al nombre de nodes emprats. Això ho veiem molt clarament amb l'eficiència, que entre 1 i 12 nodes sempre es manté per sobre del 99% i només baixa un $2,2\%$ en el cas dels 24 nodes.

Podem afirmar, doncs, que hem obtingut una aplicació sèrie que és apta per a calibratges petits, i una aplicació paral·lela que intenta solucionar els problemes de temps quan ens trobem davant l'explosió combinatòria del nombre d'escenaris a estudiar.
A més a més, constatem que hem dissenyat i desenvolupat aquesta aplicació paral·lela de forma que és altament eficient i sembla que presenta un perfil d'escalabilitat excel·lent.

Treball Futur

Tot i els bons resultats obtinguts en la paral·lelització de l'aplicació, la solució proposada encara presenta limitacions. A mesura que la mida del problema va creixent, ens continuarem trobant mancats de recursos suficients per a avaluar tots els escenaris en un temps raonable.

A més a més, el fet de presentar un sistema eficient per al calibratge de paràmetres d'un model, fa que els ecòlegs també valorin l'opció d'augmentar el nombre de paràmetres a calibrar, degut el menor cost que això representa amb l'aplicació paral·lela. Això fa que cada vegada hi hagi més demanda d'un sistema de calibratge encara més eficient.

En aquest sentit, estem treballant en dues direccions. La primera és comprovar si l'aplicació desenvolupada es pot escalar a sistemes de *High Performance Computing* (`HPC`). Per a comprovar aquesta teoria, s'ha sol·licitat accés a una instal·lació d'`HPC` del projecte HPC-Europa2 [20].

La segona opció, i a la que estem dedicant més esforços de la nostra investigació, és la d'incorporar tècniques d'Intel·ligència Artificial al nostre sistema. Es tracta de convertir aquest problema de cerca exhaustiva per un espai de solucions (que està format per els errors que cometen els escenaris) a un problema de cerca d'un mínim -local o absolut- suficientment bo per a assolir els requeriments imposats per els ecòlegs.

Esperem que aquestes investigacions ens permetin aportar un sistema de calibratge automàtic encara més eficient, que no hagi d'explorar tots els possibles escenaris, sinó que només passi per aquells que el portaran a una millor solució i que, a més a més, ho faci emprant recursos paral·lels.

Bibliografia

[1] `P-Lingua Core` http://www.p-lingua.org/wiki/index.php/PLinguaCore (Funcionant a dia 12/06/2012)

[2] `P-Lingua` http://www.p-lingua.org/wiki/index.php/Main_Page (Funcionant a dia 12/06/2012)

[3] I. Perez-Hurtado, L. Valencia, M.J. Perez-Jimenez, M.A. Colomer, A. Riscos-Núñez., *MeCoSim: A general purpose software tool for simulating biological phenomena by means of P Systems.*, Proceedings 2010 IEEE Fifth International Conference on Bio-inpired Computing: Theories and Applications (BIC-TA 2010), IEEE Press, 2010.

[4] Cardona M., Colomer M.A., Pérez-Jiménez M.J., Sanuy D., Margalida A., *Modelling ecosystems using P Systems: The Bearded Vulture, a case study.*, Lecture Notes in Computer Science 5391, 137-156, 2009.

[5] Colomer, M.A., Margalida, A. & Sanuy, D. & Pérez-Jiménez, M.J., *A bio-inspired computing model as a new tool for modeling ecosystems: the avian scavengers as a case study.*, Ecological Modelling 222, 33-47, 2011.

[6] Cardona, M., Colomer M.A., Margalida A., Pérez-Hurtado I., Pérez-Jiménez M.J., Sanuy D., *A P-System based model of an ecosystem of some scavenger birds.*, Lecture Notes in Computer Science 5957, 182-195, 2010.

[7] Margalida, A., Colomer, M.A. & Sanuy, D., *Can wild ungulate carcasses provide enough biomass to maintain avian scavenger populations? An empirical assessment using a bio-inspired computational model.*, PLoS One 6, e20248, 2011.

[8] Colomer M.A., Montoti A., García E. & Fondevilla C., *A computational model to explain annual fluctuations and extinction risk due to climate change related waterflow in a Calotriton asper population.*, EPIC - Environment & Pyrenees International Conference, Universidad de Navarra, 2011.

[9] Wikipedia - Ecosystem Model http://en.wikipedia.org/wiki/Ecosystem_model (Funcionant a dia 12/06/2012)

[10] Păun G., *Computing with membranes.*, Journal of Computer Systems Science 61, 108-143, 1998.

[11] Păun G., *A guide to membrane computing.*, Theoretical Computer Science 287, 73–100, 2002.

[12] Păun, G. & Rozenberg, G., *The Oxford Handbook of Membrane Computing.*, Oxford University Press, 2010.

[13] Pérez M. de J., Sancho F., *Computación celular con membranas: un modelo no convencional.*, Editorial Kronos, 2002.

[14] Cardona M., *Una variante de sistemas P para el modelado de ecosistemas.*, Tesis Doctoral, Universitat de Lleida, 2010.

[15] Python.org http://www.python.org/ (Funcionant a dia 12/06/2012)

[16] `Python` Speed http://wiki.python.org/moin/PythonSpeed/ (Funcionant a dia 12/06/2012)

[17] Soler, R., Agraz A., Acín J.I., Garcia M., Josa I., Sentís J.M., Cores F., *Computación de altas prestaciones en redes P2P.*, Actas de las XXI Jornadas de Paralelismo, 2010.

[18] `SQLite` http://www.sqlite.org/ (Funcionant a dia 12/06/2012)

[19] Montori A, Richter-Boix A, Franch M, Llorente GA, *Decline or natural fluctuations? 20 years survey of Calotriton asper (Dugès, 1852) population.* Basic and Applied Herpetology, 2012

[20] HPC-Europa2 http://www.hpc-europa.eu/ (Funcionant a dia 12/06/2012)

[21] Jose M. Cecilia, José M. García, Ginés D. Guerrero, Miguel A. Martínez-del-Amor, Ignacio Pérez-Hurtado, Mario J. Pérez-Jiménez., *Simulation of P systems with active membranes on CUDA.*, Briefings in Bioinformatics, 11, 3 (2010), 313-322.

Apèndix A

Article Jornadas Paralelismo 2012

A continuació trobarem l'article presentat a las XXIII Jornadas de Paralelismo organitzat per la Sociedad de Arquitectura y Tecnología de Computadores (SARTECO). Aquest article ha estat acceptat i serà exposat del 19 al 21 de Setembre de 2012 a Elx. Serà publicat a les actes d'aquestes jornades.

Paralelización de un algoritmo de calibrado para el modelado de ecosistemas

Albert Agraz[1], Josep Lluís Lérida[1], Francesc Solsona[1], Mari Angels Colomer[2]

Resumen— **Este artículo explica el proceso de paralelización de una aplicación que permite realizar de forma automática el calibrado de modelos de ecosistemas diseñados con `P-Systems`.**

Primero realizamos una introducción a los `P-Systems` y su funcionamiento. Analizaremos la imposibilidad de determinar el valor de ciertos parámetros que no son observables en la naturaleza y la necesidad de determinar unos valores adecuados para estos parámetros.

Veremos como nos encontramos delante de un problema de explosión combinatoria y calcularemos el orden de complejidad del mismo. Diseñaremos una aplicación serie que explore todas los posibles valores asignables a estos parámetros y que encuentre cuál es la que proporciona resultados más próximos a la realidad.

Acabaremos realizando una aplicación paralela que reduce sustancialmente el tiempo de ejecución del proceso de calibrado. Finalmente, indicaremos los resultados obtenidos y el trabajo futuro a realizar.

Palabras clave— **Ecosistemas, paralelización, `P-Systems`, `P-Lingua`, calibrado, explosión combinatoria, colas SGE.**

I. Introducción

La simulación matemática de procesos naturales tiene especial interés para poder conocer sus mecanismos de funcionamiento y poder predecir comportamientos bajo diferentes escenarios plausibles.

La metodología de modelado ha evolucionado a medida que el conocimiento de la realidad ha aumentado así como las prestaciones de los sistemas informáticos existentes en la actualidad. La existencia de potentes centros de cálculo, accesibles desde Internet, ha provocado un cambio en el modo de trabajar, de abordar los problemas y en el tipo de herramientas de modelado que se utilizan.

En la actualidad debemos dejar de asociar el concepto de modelado a la existencia de expresiones matemáticas que, de manera más o menos hábil, relacionan unos *inputs* con unos *outputs*. Durante muchos años se han utilizado modelos basados en las ecuaciones de Lotka Volterra [1] con muy buenos resultados, pero con limitaciones importantes como el número de procesos a modelar. En la actualidad se ha producido una explosión de modelos computacionales entre los que destacamos los modelos de viabilidad y los multiagentes. La última generación de modelos son conceptualmente más sencillos de entender pero más difíciles de describir, ya que no se pueden expresar mediante formulas analíticas.

[1]Dept. d'Informàtica i Enginyeria Industrial, Univ. de Lleida, email: albert.agraz@udl.cat, jlerida,francesc@diei.udl.es

[2]Dept. de Matemàtica, Univ. de Lleida, colomer@matematica.udl.cat

Entre los modelos computacionales de última generación están los `P-Systems` descritos por George Păun [2][3]. Son modelos bioinspirados basados en el funcionamiento de las células. Las células son organismos diminutos con unas zonas diferenciadas en su interior, que contienen una serie de orgánulos que operan simultáneamente de manera aleatoria, comunicándose e intercambiando materiales con el medio.

Los componentes básicos de un `P-System` son: la estructura de membranas, el alfabeto de trabajo y las reglas de evolución. Los componentes utilizados para modelar un problema concreto dependen de la estrategia seguida por el modelador; procurando siempre minimizar el coste computacional, que suele estar relacionado con el número de reglas. Existen distintas variantes de `P-Systems`, en el presente trabajo nos centraremos en la variante que se utiliza para la modelado de dinámicas poblacionales, denominados *Population Dynamic P-Systems models* (PDP).

Estos `P-Systems` han sido utilizados con resultados muy satisfactorios para el estudio del quebrantahuesos en los Pirineos catalanes [4][5][6][7], del mejillón cebra en el embalse de Ribarroja (Aragón), y otros como el rebeco, el acebo o el tritón pirenaico [8], el cual utilizaremos como base en la experimentación del presente trabajo.

Los parámetros que utiliza el modelo para un caso concreto son siempre los mismos independientemente del camino seguido para la obtención del *output*. Los parámetros son las medidas de campo realizadas por los expertos a lo largo del tiempo. El valor de algunos de estos parámetros no es puntual, sino que son intervalos, el experto es capaz de acotarlo pero no de precisar un único valor. Por lo tanto previa a la aplicación de los modelos para predecir futuros comportamientos, es necesario calibrar el modelo. Esto significa, buscar el valor de los parámetros que mejor ajusta los resultados del modelo a la realidad observada.

Considerando que el número de parámetros no precisados puede ser importante y estos además pueden interaccionar entre sí, nos encontramos ante un problema de exploración combinatoria, con un coste computacional importante. No obstante el correcto ajuste de estos modelos es de vital importancia, pues puede significar la diferencia entre un modelo válido y uno erróneo, es decir, entre obtener valores veraces o predicciones equivocadas. Así pues, se hace imprescindible el diseño de una herramienta de calibrado que permita el ajuste automático y eficiente de los parámetros del modelo PDP.

En este proyecto utilizaremos `P-Lingua` como motor de simulación para modelos PDP, desarrollado

por el Grupo de Investigación en Computación Natural [9] de la Universidad de Sevilla y publicado bajo la licencia `GNU GPL`. En trabajos previos [10][11] se ha tratado de paralelizar y optimizar el motor de simulación, pero nunca se ha abordado el problema del calibrado presentado en el presente trabajo.

II. Conceptos Previos

En esta sección describiremos los componentes principales de un `P-System`, los *inputs* necesarios para la modelado de un ecosistema y evaluaremos el orden de magnitud del proceso de calibración.

A. `P-Systems`

Un PDP está formado por un conjunto de entornos que, en el caso de los ecosistemas, suele estar asociado a distintos espacios geográficos. Dentro de cada entorno hay un `P-System` que no es más que un conjunto de **membranas** jerarquizadas con una determinada carga eléctrica (figura II-A). En el interior de estas membranas hay un conjunto de **objetos** que, en el caso de los ecosistemas, suelen estar asociados a los *inputs* del modelo (animales, comida,etc). Estos objetos evolucionan mediante las **reglas de evolución** pudiendo moverse entre membranas, salir del entorno, disolverse o generar nuevos objetos.

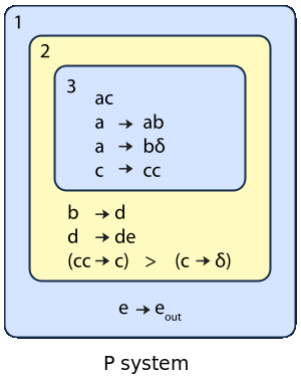

Fig. 1. Esquema de un P-Systema.

En los modelos PDP las reglas de evolución que pueden aplicarse en la región comprendida entre las membranas son del tipo:

$$r \equiv u^a[v^b]_i^\alpha \xrightarrow{f_r} u'^c[v'^d]_i^{\alpha'}$$

Si exterior a la membrana i que posee carga α, se encuentra el objeto u con multiplicidad a y en la membrana etiquetada con i el objeto v con multiplicidad b, con una probabilidad f_r se aplica la regla de manera que cambia la carga de la membrana de α a α' y los objetos u y v evolucionan a objetos u' y v' con multiplicidades c y d respectivamente.

Las reglas que se aplican entre entornos son de la forma:

$$r_e \equiv (x)_{e_j} \xrightarrow{p(x,j,j_1,\cdots,j_k)} (y_1)_{e_{j_1}} \cdots (y_k)_{e_{j_k}}$$

El objeto x pasa del entorno e_j a los entornos $e_{j_1} \cdots e_{j_k}$ en el paso entre entornos, pudiendo evolucionar a objetos $y_1 \cdots y_k$, respectivamente.

El punto de partida es la configuración inicial formada por la **estructura de membranas**, las **reglas de evolución** y el **alfabeto inicial**. En cada paso de computación se aplican todas las reglas posibles obteniéndose una configuración nueva, la secuencia de configuraciones forman una simulación.

El resultado de la simulación son los valores asignados al conjunto de objetos que se encuentran en cada membrana en el último paso de simulación.

B. Modelado de ecosistemas

Definiremos un modelo como el conjunto de reglas, membranas y alfabeto de trabajo (objetos) que definen el comportamiento de un sistema. Un modelo con valores iniciales en el alfabeto de trabajo, será denominado escenario. Un modelo dispone de tantos escenarios como combinaciones de valores posibles pueden tomar sus objetos.

El proceso de modelado de ecosistemas se sustenta en 4 fases principales, representadas en la figura II-B.

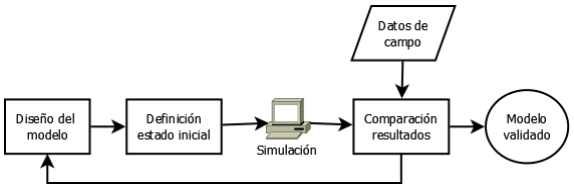

Fig. 2. Modelado de ecosistemas.

El primer paso, el **diseño del modelo**, consiste en identificar aquellas reglas de evolución que definen el comportamiento del sistema en estudio, así como la estructura de membranas que permita representar distintos estados durante la evolución, grupos de elementos, etc. Además se deben definir los objetos dentro de cada membrana. En nuestro caso, el conjunto de reglas, la estructura de las membranas y los objetos se definen mediante el lenguaje de programación `P-Lingua` [9] en un archivo fuente con extensión `.pli`.

Una vez definido el modelo, deberemos establecer el **escenario inicial**. Esto es definir un valor inicial para los objetos que hemos creado dentro de las distintas membranas para simular la evolución del sistema. La asignación de valores adecuados a estos objetos es esencial para la simulación veraz de la realidad.

Una vez definido el escenario inicial con el conjunto de reglas, estructura de membranas y valores iniciales de los objetos procedemos a la **simulación** de la evolución del sistema. Para ello utilizaremos el motor de simulación `P-LinguaCore` [12]. Esta aplicación recibe como parámetros de entrada el escenario inicial y el numero de pasos de simulación que se desean ejecutar sobre el ecosistema. El número de pasos es un parámetro definido por el ecólogo experto y está directamente relacionado con el número de años o meses que se desea evolucionar el ecosistema. Una vez finalizada la simulación obtenemos un fichero donde se registra el estado final del ecosistema y el proceso de evolución sufrido.

Dado que en el modelo definido se introduce la aleatoriedad propia de los sistemas naturales, los usuarios ecólogos tratan de simular el mismo escenario múltiples veces construyendo el resultado final como el promedio de los resultados obtenidos en las distintas simulaciones realizadas.

El modelo PDP analizado en este trabajo, modela el ecosistema del tritón pirenaico [8] mediante un único entorno con unas 20 membranas, 300 variables y 70 reglas de evolución. El coste temporal en la simulación de este modelo en una máquina convencional es aproximadamente del orden de 1 minuto y medio. Si tenemos en cuenta que se suelen hacer del orden de 30 a 100 simulaciones para normalizar los resultados nos encontramos con ejecuciones que pueden tardar de 45 minutos a las 2 horas y media.

En la última etapa del proceso de modelado debemos realizar la **validación**. Esto consiste en comprobar que los resultados del modelo se aproximan al comportamiento real del ecosistema. Para ello comparamos los resultados obtenidos a lo largo de los distintos pasos de simulación con observaciones reales del medio y calculamos el error producido. Daremos por buenos aquellos escenarios que más se aproximan a los valores observados reales.

C. Proceso de calibrado

Para obtener un buen ajuste del modelo es muy importante definir correctamente los valores de los parámetros de entrada. El valor de algunos de estos parámetros no es puntual, el experto es capaz de acotarlo pero no de precisar un valor exacto. Por lo tanto, es necesario poder ajustar con la mayor precisión posible el valor de estos parámetros. Para ello el usuario ecólogo debe generar múltiples escenarios con distintos valores para los parámetros de entrada, simular cada escenario varias veces para normalizar los resultados y comparar los resultados de cada escenario con los valores reales observados con el fin de obtener el escenario que mejor se acerca al comportamiento real.

En ocasiones el número de parámetros que el usuario ecólogo no puede precisar puede ser muy grande y además estos suelen interaccionar entre sí, por lo que nos encontramos ante un problema combinacional con un coste computacional importante que en ocasiones es inabordable. Por ejemplo, suponiendo que disponemos de 3 parámetros que pueden tomar 5 posibles valores, podemos generar hasta 5^3 posibles escenarios. En el caso de utilizar solamente 30 simulaciones para normalizar los resultados y suponiendo 1 minuto y medio para cada simulación, requeriríamos ($5^3 * 30 * 1,5 \frac{min}{sim} = 5625 min.$) aproximadamente 4 días para ajustar estos parámetros a los valores que mejores resultados proporcionan.

Dado el coste computacional tan elevado que puede supone el calibrado, se hace necesario implementar soluciones paralelas que permitan calibrar los modelos PDP para muchas variables en tiempos mucho más razonables para los usuario ecólogos. En el presente trabajo hemos implementado una solución serie con el fin de analizar el coste temporal del calibrado de un modelo PDP real usando un sistema de cómputo convencional. A continuación proponemos una solución paralela que, aún explorando todos los posibles escenarios optimiza al máximo el uso de los recursos del sistema minimizando enormemente el tiempo de obtención de la soluciones y permitiendo escalar el problema aumentando el número de parámetros a calibrar.

III. Calibración Serie

El objetivo de la aplicación serie es calibrar el modelo simulando todos los posibles escenarios un número de veces (N) determinado por el usuario ecólogo, promediando los resultados de cada uno de ellos y determinando cuál de ellos se ajusta mejor a la realidad.

Para llevar a cabo esta operación la aplicación ejecuta el procedimiento descrito en el Algoritmo 1. Como se puede ver, el algoritmo recibe como parámetros: el archivo con extensión `.pli` donde se definen la estructura de membranas y el conjunto de reglas de evolución, el archivo con extensión `.var` donde se define un valor inicial para aquellos parámetros que se van a mantener fijos en todos los escenarios y, finalmente, un archivo con extensión `.cal` donde se especifican los parámetros que deseamos calibrar y el rango de valores que pueden tomar estos parámetros, tal como se muestra a continuación:

```
q{1}=0.50:0.70:0.05
```

Donde `q{1}` es la variable a calibrar, seguida de el valor mínimo que puede tomar, el máximo y, por último, el incremento en que se desea recorrer el rango.

Algorithm 1 Calibración Serie

Require: ficheros: pli, var y cal
1: $lst\_escenarios$:= generar_lst_escenarios(cal);
2: **for** $Id\_escenario$ in $lst\_escenarios$ **do**
3: $escenario$:= generar_escenario($Id\_escenario$,pli,var);
4: **for** i in N **do**
5: $resultados[i]$:= simular($escenario$);
6: **end for**
7: $resultado$:= promediar($resultados$);
8: $error$:= distancia_euclidea($resultado$,$real$);
9: **if** $error < error_{min}$ **then**
10: $error_{min}$:= $error$;
11: $escenario_{min}$:= $escenario$;
12: **end if**
13: **end for**
14: **return** $escenario$

Una vez se dispone de los *inputs*, el algoritmo genera mediante la función `generar_lst_escenarios(cal)`, una lista ordenada con los identificadores del conjunto de escenarios posibles a simular (linea 1). Para identificar cada escenario se utiliza una lista de longitud p, donde

p representa el número de parámetros a calibrar definidos por el archivo `.cal`.

Supongamos que la tabla I representa los rangos de valores definidos por el archivo `.cal`. Cada escenario se forma como una combinación distinta de los valores de cada parámetro. Para identificar los escenarios utilizaremos una lista de longitud 5, donde cada valor i de la lista representa el índice (columna) en el rango de posibles valores del valor escogido para ese escenario. Por ejemplo, el escenario `m{3,1}=0.05, m{4,1}=0.05, m{5,1}=0.05, p{1}=0.55, p{2}=0.55` tendrá como identificador el vector `(0,0,0,0,0)`, mientras que el escenario `m{3,1}=0.08, m{4,1}=0.07, m{5,1}=0.07, p{1}=0.55, p{2}=0.10` tendrá por identificador el vector `(3,1,2,0,5)`.

La ventaja principal de esta convención es la posibilidad de referirnos a un escenario sin tener que indicar los objetos y sus valores.

TABLA I
TABLA DE CALIBRADO

Nombre	Valores					
m{3,1}	0.05	0.06	0.07	0.08	0.09	0.10
m{4,1}	0.05	0.07	0.09	0.11	0.13	
m{5,1}	0.05	0.06	0.07	0.08	0.09	0.10
p{1}	0.55	0.60	0.65			
p{2}	0.05	0.06	0.07	0.08	0.09	0.10

Una vez generada la lista de escenarios el algoritmo explora la lista para procesar cada uno de ellos (linea 2). Con el identificador del escenario que se desea procesar, el algoritmo genera, mediante la función `generar_escenario(Id_escenario,pli,var)`, un archivo compuesto por el conjunto de reglas y estructura de las membranas (`.pli`), el conjunto de valores de los parámetros fijos (`.var`) y por último el conjunto de valores del escenario a procesar (`Id_escenario`) (linea 3).

A continuación, el escenario a procesar se simula un número N de veces determinado por el usuario ecólogo con el fin de promediar los resultados (lineas 4-7). Una vez promediados los resultados se evalúa la calidad de los parámetros del escenario mediante la distancia euclídea entre los datos obtenidos fruto de la simulación y los obtenidos de las observaciones reales del medio. Llamaremos a esta distancia el "error" del escenario, y nos quedaremos con aquél cuya distancia a la observaciones reales sea mínima (lineas 8-12).

La complejidad de este algoritmo viene dada por el número de parámetros p a calibrar. Cada nuevo parámetro que añadimos hace que el número de combinaciones se multiplique por el rango de posibles valores r. En el caso en que todos los parámetros tuviesen la misma longitud en el rango de valores el total de escenarios sería r^p, cada uno de ellos simulado N veces. Dado que el total de simulaciones es constante y muy inferior al total de escenarios posibles podemos determinar que el orden de complejidad del algoritmo serie es exponencial $O(r^p)$.

IV. Calibración Paralela

Dado el grado de complejidad del algoritmo serie y la importancia para el usuario ecólogo de la calibración de estos modelos. Se hace imprescindible buscar una solución al problema de la calibración aprovechando al máximo la cantidad de recursos computacionales que nos pueden ofrecer en la actualidad entornos de Supercomputación, o entornos distribuidos *Cluster* y *Grid*. Para ello, presentamos una solución paralela a la calibración que trata de minimizar la dependencia entre las tareas paralelas, minimizar las comunicaciones entre ellas y maximizar el aprovechamiento de los recursos disponibles.

Primero debemos identificar aquellos puntos susceptibles de ser paralelizados:

- Distribución de los escenarios: consiste en simular distintos escenarios simultáneamente. Dado que los escenarios no tienen dependencia entre ellos, se pueden ejecutar tantos a la vez como nodos tengamos disponibles.
- Ejecución de las repeticiones: para mitigar los efectos de la aleatoriedad propia de los `P-Systems` simulamos cada escenario un numero determinado de veces y al final promediamos los resultados. Podríamos simular cada una de estas veces en un nodo distinto de forma simultánea.

Ambas opciones de paralelización se podrían aplicar de forma conjunta. En el presente trabajo nos hemos centrado en el paralelismo a nivel de escenario, aunque hemos preparado la aplicación para futuras incorporaciones del paralelismo a nivel de repetición.

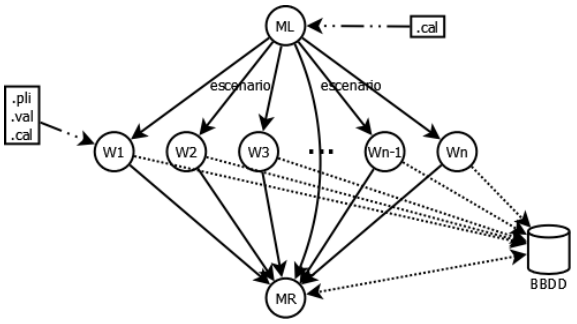

Fig. 3. Esquema de calibración paralelo.

La aplicación seguirá el paradigma Master-Worker (figura IV). Uno de los protocolos de comunicación y sincronización más utilizados para la implementación de aplicaciones paralelas en entornos distribuidos es MPI. No obstante, el procesamiento paralelo de los distintos escenarios no requiere de comunicación entre los Workers. Por otro lado, el Master solo necesita recibir el error obtenido por cada escenario con el fin de obtener el escenario que mejor se ajusta a las observaciones reales. En este caso, almacenando los resultados de los Workers en una base de datos solo se necesita notificar la finalización de cada Worker, de modo que cuando todos han finalizado el Master puede procesar los resultados. Este mecanismo de

sincronización se puede implementar también mediante las funcionalidades del sistema de colas. Por este motivo hemos decidido optar por una implementación basada en el sistema de colas del SGE. De este modo, el Master lanzará al sistema de colas el procesamiento de todos los posibles escenarios y esperara a que estos finalicen. El sistema de colas gestiona donde y cuando se debe procesar cada escenario, y cuando todas acaban se relanza el Master que procesa los resultados almacenados en la base de datos para decidir el mejor ajuste de los parámetros.

El proceso paralelo implementado se divide en tres fases principales: el **despliegue** de todos los escenarios, la **ejecución paralela** de las simulaciones por parte de los Workers y, finalmente, el **procesamiento de los resultados** para obtener el mejor ajuste de los parámetros.

En el Algoritmo 2 se detallan los pasos que realiza el Master para desplegar la ejecución de todos los escenarios. En primer lugar crea la base de datos en una localización accesible tanto desde el master como los workers (linea 1). A continuación, se generan la lista ordenada con los identificadores de los escenarios que se deben simular (linea 2). Por cada escenario de la lista, mediante la función `lanzar_simulación(Id_escenario,N)`, se encola un Worker en la cola del SGE (lineas de 3 a 5). Finalmente, se relanza a sí mismo para la última fase.

Algorithm 2 Fase de despliegue - Master

Require: fichero: cal
1: crear_base_datos(*localizacion*);
2: *lst_escenarios* := generar_lst_escenarios(*cal*);
3: **for** *escenario* in *lst_escenarios* **do**
4: lanzar_simulacion(*Id_escenario*, *N*);
5: **end for**
6: relanzar_master();

La **fase de ejecución** la realiza el Worker. Como podemos observar en el Algoritmo 3, el Worker se conecta a la base de datos (linea 1). A continuación, a partir de los archivos `.pli`, `.var`, `.cal` y del identificador del escenario `Id_escenario`, se genera el archivo con todos los datos del escenario a simular (linea 2).

Algorithm 3 Fase de ejecución - Worker

Require: código, ficheros: pli, var y cal
1: conectar_base_datos();
2: *escenario* := generar_escenario(*Id_escenario*, *pli*, *var*, *cal*);
3: **for** *i* in *N* **do**
4: *resultados*[*i*] := simular(*escenario*);
5: **end for**
6: *resultado* := promediar(*resultados*);
7: *error* := distancia_euclidea(*resultado*,*real*);
8: guardar_base_datos(*Id_escenario*, *error*);

Una vez generado el archivo se simula cada escenario un numero N de veces determinado por el usuario ecólogo (lineas 3-5), se promedia el conjunto de resultados obtenidos (linea 6) y se calcula el error producido por el escenario (linea 7). Finalmente, el error se almacena en la base de datos junto con el identificador del escenario (linea 8).

Algorithm 4 Fase de recolección - Master

1: conectar_base_datos();
2: *best* = consultar_mejor_escenario();
3: **return** *best*

La última fase es la **fase de recolección**, realizada por el master y descrita en el Algoritmo 4. Una vez finalizan todos los Workers, els SGE lanza de nuevo el proceso Master, el qual se conecta a la base de datos (linea 1) y lanza una consulta SQL para encontrar el escenario que ha generado el error mínimo (linea 2). Finalmente, devuelve los valores que mejor ajustan los parámetros no fijados por el usuario ecólogo.

V. Experimentación

El objetivo principal de esta experimentación es evaluar el coste computacional del problema de la calibración así como analizar la eficiencia que se obtiene con nuestra propuesta de paralelización del algoritmo de calibración. Para realizar las pruebas de rendimiento hemos usado un sistema PDP que modela el ecosistema del tritón Pirenaico [8]. Simularemos un período de 10 años y valoraremos la calidad de las soluciones comparandolas con las observaciones reales disponibles. Nos interesa evaluar y comprar los tiempos de ejecución de la aplicación serie y paralela.

La aplicación serie se ha ejecutado sobre un equipo con Intel Core 2 Quad a 2.4GHz y 4GB de RAM. La aplicación paralela se ha ejecutado sobre un *Cluster* homogéneo de 24 nodos con las mismas características que el equipo utilizado para la aplicación serie.

TABLA II
Tabla de resultados

Proc.	Tiempo (s)	Aceleración	Eficiencia
Serie	123240	-	-
1	126975	1	1
3	42420	2,993	0,9977
6	21300	5,961	0,9935
12	10620	11,956	0,9963
24	5460	23,255	0,9689

La tabla II nos muestra los tiempos de ejecución obtenidos. Se observa cómo la ejecución paralela con un único procesador es únicamente un 3% más lenta que la aplicación serie. Los valores de la eficiencia (Figura V) siempre superiores al 96% nos indican que el *Speedup* se mantiene cercano al ideal.

En la figura V podemos observar como el *Speedup* aumenta con el número de nodos de modo que la

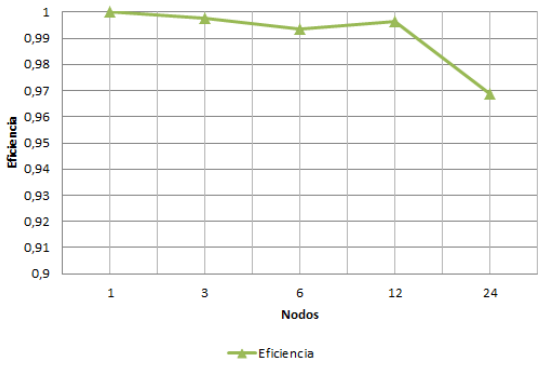

Fig. 4. Gráfico de eficiencia.

eficiencia se mantiene a medida que aumentamos el número de recursos computacionales a utilizar.

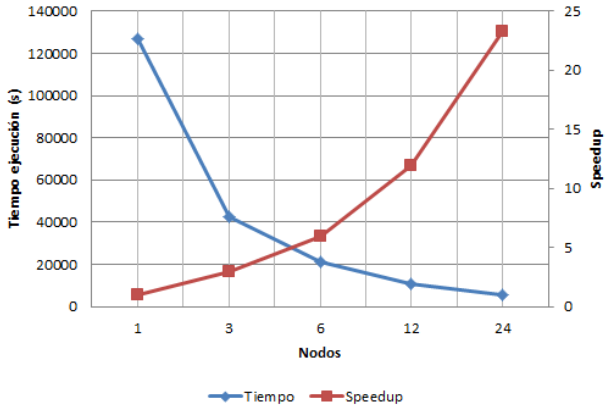

Fig. 5. Gráfico de rendimiento.

VI. Conclusiones y Trabajo Futuro

El tiempo obtenido con la ejecución serie nos muestra el alto coste computacional de la calibración. Hay que tener en cuenta que el modelo utilizado en esta experimentación es un modelo de tamaño medio-pequeño. Esto significa que para modelos superiores se aumenta la complejidad computacional del problema. Los resultados obtenidos para la ejecución paralela nos muestra como el *Speedup* mantiene un crecimiento casi lineal y la eficiencia del sistema implementado se mantiene por encima del 99% en los casos de 1 a 12 nodos, y sólo baja un $2,2\%$ en el caso de los 24 nodos. Con esta evolución podemos afirmar que se trata de una aplicación altamente escalable.

En la actualidad estamos trabajando en la incorporación de técnicas Heurísticas para la exploración del espacio de escenarios posibles. De este modo pretendemos convertir el problema de la explosión combinatoria debido a la exploración exhaustiva del espacio de soluciones, en un problema de búsqueda del mínimo global (o una aproximación lo suficientemente buena) y en un tiempo mucho más que razonable para el usuario ecólogo. Esto nos permite no solo escalar el problema, sino atacar modelos mucho más complejos y proporcionar resultados importantes para la toma de decisiones por parte de los gestores del medio. En este sentido hemos identificado y empezado a experimentar distintos algoritmos (diseño de experimentos, búsqueda del mínimo gradiente, clasificación por árboles binarios, etc.) que se están mostrando eficaces para este tipo de problemas.

Agradecimientos

El presente trabajo ha sido financiado por el Ministerio de Ciencia e Innovación mediante el contrato TIN2011-28689-C02-02. Los autores son miembros del grupo de investigación consolidado de la Generalitat de Cataluña 2009 SGR145.

Referencias

[1] Leigh, E.R., *The ecological role of Volterra's equations.*, Some Mathematical Problems in Biology, 1968.
[2] Păun, G., *Computing with membranes.*, Journal of Computer Systems Science 61, 108-143, 1998.
[3] Păun, G., Rozenberg, G. & Salomaa, A., *The Oxford Handbook of Membrane Computing.*, Oxford University Press, 2010.
[4] Cardona M., Colomer M.A., Pérez-Jiménez M.J., Sanuy D., Margalida A., *Modelling ecosystems using P Systems: The Bearded Vulture, a case study.*, Lecture Notes in Computer Science 5391, 137-156, 2009.
[5] Colomer, M.A., Margalida, A. & Sanuy, D. & Pérez-Jiménez, M.J., *A bio-inspired computing model as a new tool for modeling ecosystems: the avian scavengers as a case study.*, Ecological Modelling 222, 33-47, 2011.
[6] Cardona, M., Colomer M.A., Margalida A., Pérez-Hurtado I., Pérez-Jiménez M.J., Sanuy D., *A P-System based model of an ecosystem of some scavenger birds.*, Lecture Notes in Computer Science 5957, 182-195, 2010.
[7] Margalida, A., Colomer, M.A. & Sanuy, D., *Can wild ungulate carcasses provide enough biomass to maintain avian scavenger populations? An empirical assessment using a bio-inspired computational model.*, PLoS One 6, e20248, 2011.
[8] Colomer M.A., Montoti A., García E. & Fondevilla C., *A computational model to explain annual fluctuations and extinction risk due to climate change related water-flow in a Calotriton asper population.*, EPIC - Environment & Pyrenees International Conference, Universidad de Navarra, 2011.
[9] http://www.p-lingua.org/wiki/index.php/Main_Page
[10] I. Perez-Hurtado, L. Valencia, M.J. Perez-Jimenez, M.A. Colomer, A. Riscos-Núñez., *MeCoSim: A general purpose software tool for simulating biological phenomena by means of P Systems.*, Proceedings 2010 IEEE Fifth International Conference on Bio-inpired Computing: Theories and Applications (BIC-TA 2010), IEEE Press, 2010.
[11] Jose M. Cecilia, José M. García, Ginés D. Guerrero, Miguel A. Martínez-del-Amor, Ignacio Pérez-Hurtado, Mario J. Pérez-Jiménez., *Simulation of P systems with active membranes on CUDA.*, Briefings in Bioinformatics, 11, 3 (2010), 313-322.
[12] http://www.p-lingua.org/wiki/index.php/PLinguaCore

Apèndix B

Codi de l'Aplicació Sèrie

En el llistat B.1 mostrem el codi `Python` de l'aplicació sèrie per al calibratge automàtic de models descrita en l'apartat 4.

```
#!/usr/bin/python
# coding=<utf-8>
'''
Created on Apr 15, 2012

@author: albert
'''

import time
import os
from utils import utils
from estadistics import estadistic
from db import sqlite
import datetime
import hashlib


#### variables d'entrada ##############################################
pli_file        = open("./files/newtoctubre.pli", "r")  #
    .pli_val file descriptor
cal_file        = open("./files/newtoctubre.cal", "r")  # .cal_val
    file descriptor
var_file        = open("./files/newtoctubre.var", "r")  # .var_val
    file descriptor
especies_file   = "./files/especies.esp"                # fitxer
    amb les noms_variables a estudiar
reals_file      = "./files/escenari_ideal.avg"          # fitxer
    amb les dades reals (ideal)
plingua_path    = "./files/pLinguaCore.jar"             # path to
    plinguacore.jar file
######################################################################
path            = "/state/partition1"                   #
    directori d'execucio (local)
######################################################################
estadistic_path             = "./resultats_serie/estadistics"
          # directori de resultats (compartit)
localitzacio_base_dades     = "./resultats_serie/psyscal.db"
           # path de la base de dades (compartit)
######################################################################
steps_any       = 17            # steps per any
anys            = 20            # nombre d'anys de la simulacio
```

```
nrep            = 30            # num repeticions per simulacio
tipus_error     = "distancia"   # algoritme calcul error
##############################################################################


#### variables globals ########################################################
noms_variables      = []    # noms de les noms_variables
valors_variables    = []    # taula de calibratge
it                  = []    # iteradors per les noms_variables
pli_val             = pli_file.read()
var_val             = var_file.read()
nom_model           = os.path.basename(pli_file.name).split(".")[0]
##############################################################################




if __name__ == '__main__':

    init_time = time.time()
    print(time.ctime())

    min_error = 9999999999.999
    min_escenari = "escenari_inicial"

    conn = sqlite.setup_database(localitzacio_base_dades,
        nom_model)
    utils.prepare_directories(path)
    utils.read_cal_file(noms_variables, valors_variables,
        cal_file, it)

    #calculem nombre de combinacions
    ncomb = utils.calcul_nombre_combinacions(noms_variables,
        valors_variables)
    print("Iniciem bucle amb " + str(ncomb) + " combinacions.")

    #bucle per totes les combinacions
    for i in range(ncomb):

        #computem el nom de l'escenari
        nom_escenari = utils.obtenir_nom_escenari(nom_model, it)

        #preparem segment variables de l'escenari
        var_val = utils.aplicar_valors_iteradors(nom_model,
            noms_variables, valors_variables, it, var_val)

        #comprovem si l'escenari esta a la base de dades
        hash_escenari = hashlib.sha256(var_val).hexdigest()
        error_escenari = sqlite.comprova_escenari(nom_model,
            hash_escenari, reals_file)

        if error_escenari == None:
            print("\nEscenari: " + nom_escenari + " hash_escenari:
                " + hash_escenari + " no trobat a la DB. Simulant
                escenari...")

            #imprimir escenari .pli
            utils.print_file(nom_escenari, path, pli_val, var_val)

            #executar fitxer nrep repeticions i calcul de
                l'error_escenari
```

```
            for i in range(nrep):
                # executem plingua
                print("\trepeticio: " + str(i))
                ordre = "java -jar " + plingua_path + "
                    plingua_sim -pli " + path + "/pli/" +
                    nom_escenari + ".pli" + " -o " + path +
                    "/dat/" + nom_escenari + "_" + str(i) + ".dat"
                    + " -st " + str(steps_any*anys) + " -mode
                    dndp4 -v 0 > /dev/null"
                os.system(ordre)

            #calculem l'error_escenari
            print("\t Calculant estadistiques...")
            error_escenari = estadistic.cerror(steps_any, path +
                "/dat/", nom_escenari, reals_file, especies_file,
                estadistic_path, path, path)
            print("\t Estadistiques calculades.")

            #guardar valors_variables base de dades
            sqlite.insereix_resultat(nom_model, nom_escenari,
                hash_escenari, None, error_escenari,
                algoritme_error, reals_file,
                str(datetime.datetime.now()), None, None)

            #esborrar fitxers dat i pli
            utils.esborrar_fitxers(path, nom_escenari, nrep)

        else:
            print("Escenari: " + nom_escenari + " hash_escenari: "
                + hash_escenari + " trobat a la BD.")

        print("Escenari: " + nom_escenari + " hash_escenari: " +
            hash_escenari + " error_escenari: " +
            str(error_escenari))

        #es millor que l'escenari que tenim?
        if error_escenari < min_error:
            print("\nNou minim trobat: " + nom_escenari + "
                error_escenari: " + str(error_escenari))
            min_error = error_escenari
            min_escenari = nom_escenari

        #augmentar index
        it[0] = it[0] + 1
        #comprovar augment de l'index
        for k in range(len(noms_variables)):
            if it[k] >= len(valors_variables[k]):
                it[k] = 0
                try:    #capturem l'exepcio de l'ultim acces
                    it[k + 1] = it[k + 1] + 1
                except IndexError:
                    break
            else:
                break

    #fer neteja final de pli, dat i tot
    utils.remove_directories(path)

    #tancar base de dades
    sqlite.tanca_database()

    end_time = time.time()
```

```
    print("--- FINAL ---")
    print("Millor escenari: " + min_escenari)
    print("Error: " + str(min_error))
    print("Temps d'execucio: " + str(end_time - init_time)) + "
        segons"
    print(time.ctime())
```

Listing B.1: Codi de l'aplicació sèrie.

Apèndix C

Codi de l'Aplicació Paral·lela

En el llistat C.1 mostrem el codi corresponent als algoritmes executats pel Master en l'aplicació paral·lela. Està implementat en `Python`. En el llistat C.2 veurem el codi del Worker.

```
#!/usr/bin/python
# coding=<utf-8>
'''
Created on 14/05/2012

@author: Albert
'''

import time
import sys
import os
from utils import utils
from db import sqlite


#### variables d'entrada ##########################################
cal_file        = open("./files/newtoctubre.cal", "r") # .cal_val
    file
###################################################################
path            = "/state/partition1"   # directori d'execucio
    (local)
###################################################################
estadistic_path = "./resultats_serie/estadistics" # dir resultats
    (comp.)
localitzacio_base_dades = "./resultats_serie/psyscal.db" # path db
    (comp.)
###################################################################
error_max       = 2                     # error_escenari maxim permes
tipus_error     = "distancia"           # tipus d'error
###################################################################

#### variables globals ############################################
noms_variables      = []    # noms de les noms_variables
valors_variables    = []    # taula calibratge
it                  = []    # iteradors per noms_variables
pli_val             = pli_file.read()
```

```
var_val                 = var_file.read()
nom_model               = os.path.basename(pli_file.name).split(".")[0]
##############################################################################


if __name__ == '__main__':

    print(time.ctime())
    init_time = time.time()

    # mirem si es la segona execucio -> recollir dades
    if len(sys.argv) > 1 and sys.argv[1] == "recollir":
        #connectem base dades i obtenir minim
        sqlite.connexio_database(localitzacio_base_dades)
        res = sqlite.trobar_minim(nom_model)

        # imprimir minim i data
        print("--- FINAL ---")
        print("Nom escenari: " + res[0])
        print("Hash escenari: " + res[1])
        print("Error: " + str(res[2]))
        print("Temps execucio segon master: " +
            str(time.time()-init_time) + " segons")
        print(time.ctime())

        # tanquem base de dades
        sqlite.tanca_database()

        #tanca programa
        sys.exit()

    #creem database
    conn = sqlite.setup_database(localitzacio_base_dades,
        nom_model)

    #reconstruim variables i it
    utils.read_cal_file(noms_variables, valors_variables,
        cal_file, it)
    utils.inicialitza_it(it, noms_variables)

    #calculem nombre de combinacions
    ncomb = utils.calcul_nombre_combinacions(noms_variables,
        valors_variables)
    print("Iniciem bucle amb " + str(ncomb) + " combinacions.")

    #bucle per totes les combinacions
    for i in range(ncomb):

        #computem el nom de l'escenari
        identificador_escenari =
            utils.construir_identificador_escenari(it)

        #llensar escenari
        ordre = "ssh maracas.local 'qsub
            /home/albert/PSysCal-par/src/worker.sh " +
            identificador_escenari + " &'"
        print("executem: " + ordre + " per a id=" +
            identificador_escenari)
        os.system(ordre)

        #augmentar index
        it[0] = it[0] + 1
```

```
        #comprovar augment de l'index
        for k in range(len(noms_variables)):
            if it[k] >= len(valors_variables[k]):
                it[k] = 0
                try:    #capturem l'exepcio de l'ultim acces
                    it[k + 1] = it[k + 1] + 1
                except IndexError:
                    break
            else:
                break

    # rellancem master
    ordre = "ssh maracas.local 'qsub " + "-hold_jid worker.sh"+ "
        /home/albert/PSysCal-par/src/master.sh " + "recollir" + "
        &'"
    print("\nRellnasem el master. Executem: " + ordre)
    os.system(ordre)

    end_time = time.time()

    print("Hem rellansat master.")
    print("Temps d'execucio primer master: " + str(end_time -
        init_time)) + " segons"
```

Listing C.1: Codi de l'aplicació paral·lela - Master.

```
#!/usr/bin/python
# coding=<utf-8>
'''
Created on 14/05/2012

@author: Albert
'''

import time
import os
from utils import utils
from estadistics import estadistic
from db import sqlite
import sys
import datetime
import hashlib


#### variables d'entrada ########################################
pli_file        = open("./files/newtoctubre.pli", "r") # .pli_val
    file
cal_file        = open("./files/newtoctubre.cal", "r") # .cal_val
    file
var_file        = open("./files/newtoctubre.var", "r") # .var_val
    file
especies_file   = "./files/newtoctubre.esp"              #
    noms_variables
reals_file      = "./files/escenari_ideal.avg"         # dades
    reals
plingua_path    = "./files/pLinguaCore.jar"            #
    plinguacore.jar
#################################################################
path            = "/state/partition1"   # directori d'execucio
    (local)
#################################################################
```

```
estadistic_path = "./resultats_serie/estadistics" # dir resultats
    (comp.)
localitzacio_base_dades = "./resultats_serie/psyscal.db" # path db
    (comp.)
#########################################################################
steps_any       = 17                # steps per any
anys            = 20                # nombre d'anys de la simulacio
nrep            = 30                # numero de repeticions per
    simulacio
algoritme_error = "experiments"     # tipus alg: experiments,
    cerca,...
tipus_error     = "distancia"       # tipus d'error
#########################################################################

#### variables globals ##################################################
noms_variables      = []    # noms de les noms_variables
valors_variables    = []    # taula calibratge
it                  = []    # iteradors per noms_variables
pli_val             = pli_file.read()
var_val             = var_file.read()
nom_model           = os.path.basename(pli_file.name).split(".")[0]
#########################################################################


if __name__ == '__main__':

    print(time.ctime())
    init_time = time.time()

    #utils.read_params(path)
    it = sys.argv[1].split("_")
    for i in range(len(it)):
        it[i] = int(it[i])
    utils.prepare_directories(path)
    utils.read_cal_file(noms_variables, valors_variables,
        cal_file, it)

    #computem el nom de l'escenari
    nom_escenari = utils.obtenir_nom_escenari(nom_model, it)

    #preparem segment variables de l'escenari
    var_val = utils.aplicar_valors_iteradors(nom_model,
        noms_variables, valors_variables, it, var_val)

    #conectem amb la base de dades
    sqlite.connexio_database(localitzacio_base_dades)

    #comprovem si l'escenari esta a la base de dades
    hash_escenari = hashlib.sha256(var_val).hexdigest()
    error_escenari = sqlite.comprova_escenari(nom_model,
        hash_escenari, reals_file)


    if error_escenari == None:
        print("Escenari: " + nom_escenari + " hash_escenari: " +
            hash_escenari + " no trobat a la DB. Simulant
            escenari...")

        #imprimir escenari .pli
        utils.print_file(nom_escenari, path, pli_val, var_val)
```

```
        #executar fitxer nrep repeticions i calcul de
            l'error_escenari
        for i in range(nrep):
            # executem plingua
            print("\t repeticio: " + str(i))
            ordre = "java -jar " + plingua_path + " plingua_sim
                -pli " + path + "/pli/" + nom_escenari + ".pli" +
                " -o " + path + "/dat/" + nom_escenari + "_" +
                str(i) + ".dat" + " -st " + str(steps_any*anys) +
                " -mode dndp4 -v 0 > /dev/null "
            os.system(ordre)

        #calculem l'error_escenari
        print("\t Calculant estadistiques...")
        error_escenari = estadistic.cerror(steps_any, path +
            "/dat/", nom_escenari, reals_file, especies_file,
            estadistic_path, path, path)
        print("\t Estadistiques calculades.")

        #guardar valors_variables base de dades
        print("Inserim a la base de dades: escenari: " +
            nom_escenari + " hash_escenari: " + hash_escenari + "
            error: " + str(error_escenari))
        sqlite.insereix_resultat(nom_model, nom_escenari,
            hash_escenari, None, error_escenari, algoritme_error,
            reals_file, str(datetime.datetime.now()), None, None)

        #esborrar fitxers
        utils.esborrar_fitxers(path, nom_escenari, nrep)
        utils.remove_directories(path)

        #tanca base de dades
        sqlite.tanca_database()

    print("--- FINAL ---")
    print("Escenari: " + nom_escenari)
    print("Hash: " + hash_escenari)
    print("Error: " + str(error_escenari))
    print("Temps d'execucio: " + str(time.time() - init_time)) + "
        segons"
    print(time.ctime())
```

Listing C.2: Codi de l'aplicació paral·lela - Worker.

Apèndix D

El Model del Tritó Pirinenc

Com ja hem vist al llarg d'aquesta memòria, el model del Tritó pirinenc ens ha servit de joc de proves per a testejar la nostra aplicació. A continuació, volem mostrar el contingut dels fitxers involucrats en aquest procés.

El primer fitxer és el `.pli` que defineix les regles del model, el trobem al llistat D.1. El següent fitxer a tenir en compte és el `.var` amb la definició de l'alfabet de treball i els valors inicials d'aquest alfabet. Recordem que els valors inicials de les variables a calibrar seran ignorats. Trobem aquestes definicions en el llistat D.2.

També caldrà tenir en compte el fitxer `.cal` per a definir les variables que calibrarem i els possibles valors que poden prendre aquestes. Aquest fitxer es troba reproduït en el llistat D.3. Afegirem el fitxer `.esp` que delimita els elements de l'alfabet que estudiarem (veure llistat D.4.

Finalment, necessitarem les dades reals de les variables indicades en el fitxer `.esp`. Aquestes dades les trobarem en un fitxer `.avg` (veure llistat D.5).

```
@model<probabilistic>
log_e = @log(2.718281828459);
def newt_ampliat(N,E,NY)
/* Calotriton asper */
{
  /* Membrane structure */

  @mu=    [
  [ [[ [ ]'11    ]'1 [ [ ]'21    ]'2 [ [ ]'31    ]'3 [ [ ]'41    ]'4 [
       [ ]'51    ]'5 [ [ ]'61    ]'6 [ [ ]'71    ]'7 [ [ ]'81    ]'8 [
       [ ]'91    ]'9]'0 ]'111,111   ]'p;

/* Initial multisets */
  @ms({k},111) += X{1,j}*(q{j}*(qc{k}*0.25)):
      1<=k<=E,g{2,i}<=j<=g{5,i},1<=i<=2;
  @ms({k},111) +=R{0}:1<=k<=E;
  @ms({k},111) +=RI{i}:1<=k<=E, 1<=i<=N;
    @ms(0,111) +=F{1,k}:1<=k<=E;


/***************** Reintroduccion ******************/


  /*r1*/ [RI{i}--> a{i}*25000, e{i}*2000,
      Yp{i,g{5,i}/2}*NR{i,k}]'{k} ::1: 1<=i<=N, 1<=k<=E;
```

```
  /*r2*/[e{i}--> a{i}]'{k} ::0.5: 1<=i<=N,1<=k<=E;
  /*r3*/[e{i}--> #]'{k} ::0.5: 1<=i<=N,1<=k<=E;

/***************** Simulate flood module ******************/

  /*r4*/[F{i,k} --> Sp{k,I{i}}, Fp{i,k}]'0 :: 1:1<=i<=NY, 1<=k<=E;
  /*r5*/ Sp{k,1}[]'{k} --> [S{1},N]'{k}::1:1<=k<=E;
  /*r6*/ Sp{k,2}[]'{k} --> [S{2}]'{k}::1:1<=k<=E;
  /*r7*/ Sp{k,3}[]'{k} --> [S{1}, S{2}]'{k}::1:1<=k<=E;
  /*r8*/ Sp{k,0}[]'{k} --> [N]'{k}::1:1<=k<=E;

  /*******************************

  /*r9*/ [R{i} --> R{i+1}]'{k} :: 1: 0<=i<= 11, i<>4,
       i<>6,i<>9,1<=k<=E;
  /*r10*/ R{4} []'{k*10+1} -->R{5}+[#]'{k*10+1}::1: 1<=k<=E;
  /*r11*/ R{6} +[]'{k*10+1} -->R{7}[#]'{k*10+1}::1: 1<=k<=E;

  /*r12*/ R{9}[]'{k*10+1} -->R{10}+[#]'{k*10+1}::1: 1<=k<=E;
  /*r13*/ R{12}, S{2} []'{k*10+1} --> R{13}-[#]'{k*10+1}
       ::1:1<=k<=E;
  /*r14*/ R{12}, N []'{k*10+1} --> R{13}[#]'{k*10+1} ::1:1<=k<=E;
  /*r15*/ R{13} -[]'{k*10+1} --> R{14}[#]'{k*10+1} ::1:1<=k<=E;
  /*r16*/ R{13} []'{k*10+1} --> R{14}[#]'{k*10+1} ::1:1<=k<=E;


  /*r17*/ [R{14}]'{k}--> -[R{15}]'{k} ::1:1<=k<=E;
  /*r18*/ -[R{15}]'{k} --> YR +[R{16}]'{k} ::1:1<=k<=E;
  /*r19*/ +[R{16}]'{k} --> [RI{1},RI{2},R{0}]'{k} ::1: 1<=k<=E;
  /*r20*/ [YR, Fp{i,k} --> F{i+1,k}]'0 ::1: 1<=i<NY, 1<=k<=E;


/***************óinundacin primavera*********************/
  /*r21*/ S{1}[]'{k*10+1} --> -[S{1}]'{k*10+1}::1:1<=k<=E;
  /*r22*/ -[S{1}]'{k*10+1} --> [ ]'{k*10+1}::1:1<=k<=E;

  /*r23*/ X{i,j}-[]'{10*k+1} --> [X{i,j}]'{10*k+1} :: 1:
       1<=j<=g{5,i},1<=i<=N, 1<=k<=E;
  /*r24*/ [X{i,1}]'{10*k+1} -->+[#]'{10*k+1} :: mf{1,i,1}:
       1<=i<=N, 1<=k<=E;
  /*r25*/ [X{i,1}]'{10*k+1} -->X{1,i}+[]'{10*k+1} :: 1-mf{1,i,1}:
       1<=i<=N, 1<=k<=E;
  /*r26*/ [X{i,j}]'{10*k+1} --> +[]'{10*k+1} :: mf{2,i,1}:
       g{2,i}+1<=j<=g{3,i}+1,1<=i<=N, 1<=k<=E;
  /*r27*/ [X{i,j}]'{10*k+1} --> X{i,j}+[]'{10*k+1} ::
       1-mf{2,i,1}: g{2,i}+1<=j<=g{3,i}+1,1<=i<=N, 1<=k<=E;
  /*r28*/ [X{i,j}]'{10*k+1} --> +[]'{10*k+1} :: mf{3,i,1}:
       g{3,i}+1<j<=g{4,i}+1,1<=i<=N, 1<=k<=E;
  /*r29*/ [X{i,j}]'{10*k+1} --> X{i,j}+[]'{10*k+1} :: 1-mf{3,i,1}:
       g{3,i}+1<j<=g{4,i}+1,1<=i<=N, 1<=k<=E;
  /*r30*/ [X{i,j}]'{10*k+1} --> +[]'{10*k+1} :: mf{4,i,1}:
       g{4,i}+1<j<=g{5,i},1<=i<=N, 1<=k<=E;
  /*r31*/ [X{i,j}]'{10*k+1} --> X{i,j}+[]'{10*k+1} ::
       1-mf{4,i,1}: g{4,i}+1<j<=g{5,i},1<=i<=N, 1<=k<=E;

  /******************* Reproduction
       module*****************************/

/*edat de óreproducci*/
    /*r32*/ X{i,j}+[]'{10*k+1}--> +[X{i,j}]'{10*k+1} ::
        (1-g{1,i}): g{6,i}<=j<=g{7,i},1<=i<=2, 1<=k<=E;
```

```
    /*r33*/ X{1,j} +[]'{10*k+1}-->+[
        X{1,j},L{1}*(29.906*(@log(j)/log_e)-35.221)]'{10*k+1} ::
        g{1,1}*g{8,1}: g{6,1}<=j<=g{7,1},1<=k<=E;

  /*r34*/ X{2,j}+[]'{10*k+1}--> +[X{2,j},L{2}*(30)]'{10*k+1} ::
      g{1,2}*g{8,2}: g{6,2}<=j<=g{7,2},1<=k<=E;

  /*r35*/ X{i,j}+[]'{10*k+1}--> +[X{i,j}]'{10*k+1} ::
      g{1,i}*(1-g{8,i}): g{6,i}<=j<=g{7,i},1<=i<=2, 1<=k<=E;

/* grans que ja no es reprodueixen**/
   /*r36*/ X{i,j}+[]'{10*k+1}--> +[X{i,j}]'{10*k+1} ::
       1:g{7,i}<j<=g{5,i},1<=i<=2,1<=k<=E;

/*** joves que no tenen edat per reproduir-se*/

  /*r37*/ X{i,j}+[]'{10*k+1}--> +[X{i,j}]'{10*k+1} :: 1:
      g{2,i}<=j<g{6,i}, 1<=i<=2, 1<=k<=E;
  /*********************************/
  /*r38*/ Yp{i,j}+[]'{10*k+1}--> +[Yp{i,j}]'{10*k+1} :: 1:
      1<=j<=g{5,i},1<=i<=2, 1<=k<=E;

  /*r39*/ +[X{i,j}]'{10*k+1}--> Y{i,j}[]'{10*k+1} :: 1:
      1<=j<=g{5,i}, 1<=i<=2, 1<=k<=E;
  /*r40*/ +[L{i}]'{10*k+1}--> L{i}[]'{10*k+1} :: 1:  1<=i<=2,
      1<=k<=E;

  /*r41*/ +[Yp{i,j}]'{10*k+1}--> Y{i,j}[]'{10*k+1} :: 1:
      1<=j<g{5,i}, 1<=i<=2, 1<=k<=E;

/********************  Feeding module*****************************/

  /*r42*/ Y{2,j}[]'{k*10+1} --> Z{2,j}-[a]'{k*10+1} :: 1:
      g{2,2}<=j<=g{5,2}, 1<=k<=E;
  /*r43*/ L{1}-[]'{k*10+1} --> [L{1}]'{k*10+1} :: 1: 1<=k<=E;
  /*r44*/ Y{1,j}-[]'{k*10+1} --> [Y{1,j}]'{k*10+1} ::
      1:g{2,1}<=j<=g{5,1}, 1<=k<=E;
  /*r45*/ -[a]'{k*10+1} --> [#]'{k*10+1} :: 1: 1<=k<=E;

/*********** larvae *****************************/

  /*r46*/ [L{1}]'{k*10+1} --> +[#]'{k*10+1} :: p{1}: 1<=k<=E;
  /*r47*/ [L{1}]'{k*10+1} --> L{1}+[]'{k*10+1} :: 1-p{1}: 1<=k<=E;

/********** young animals ************************/
  /*r48*/ [Y{1,j}]'{k*10+1} --> +[#]'{k*10+1} ::
      p{2}:g{2,1}<=j<=g{3,1}, 1<=k<=E;
  /*r49*/ [Y{1,j}]'{k*10+1} --> Z{1,j}+[]'{k*10+1} ::
      1-p{2}:g{2,1}<=j<=g{3,1}, 1<=k<=E;
  /*r50*/ [Y{1,j}]'{k*10+1} --> Z{1,j}+[#]'{k*10+1} ::
      1:g{3,1}<j<=g{5,1}, 1<=k<=E;

/********************  Natural mortality
    module*****************************/

  /*********fase larvaria******/
  /*r51*/ L{i}+[]'{k*10+1}--> [#]'{k*10+1} ::
      (m{1,i}*(1-m{2,i})+m{2,i}): 1<=i<=N, 1<=k<=E;
  /*r52*/ L{i}+[]'{k*10+1}--> [Lpp{i}]'{k*10+1} ::
      1-(m{1,i}*(1-m{2,i})+m{2,i}):1<=i<=N, 1<=k<=E;
  /*****************Joves aigua terra *****/
  /*r53*/ Z{i,j}+[]'{k*10+1}--> [#]'{k*10+1} :: m{3,i}:
```

```
      g{2,i}<=j<=g{3,i},1<=i<=N, 1<=k<=E;
  /*r54*/ Z{i,j}+[]'{k*10+1}--> [Wp{i,j}]'{k*10+1} :: 1-m{3,i}:
      g{2,i}<=j<=g{3,i},1<=i<=N,  1<=k<=E;
  /********************** Adults etapa 1  aigua
      terra****************/
  /*r55*/ Z{i,j}+[]'{k*10+1}--> [#]'{k*10+1} :: m{4,i}:
      g{3,i}<j<=g{4,i},1<=i<=N,  1<=k<=E;
  /*r56*/ Z{i,j}+[]'{k*10+1}--> [Wp{i,j}]'{k*10+1} :: 1-m{4,i}:
      g{3,i}<j<=g{4,i},1<=i<=N,  1<=k<=E;
  /****************Adults etapa 2*****/
  /*r57*/ Z{i,j}+[]'{k*10+1}--> [#]'{k*10+1} :: m{5,i}:
      g{4,i}<j<g{5,i},1<=i<=N,  1<=k<=E;
  /*r58*/ Z{i,j}+[]'{k*10+1}--> [Wp{i,j}]'{k*10+1} :: 1-m{5,i}:
      g{4,i}<j<g{5,i},1<=i<=N,  1<=k<=E;
  /************* çesperana vida *********/
  /*r59*/ Z{i,g{5,i}}+[]'{k*10+1}--> [#]'{k*10+1}::1: 1<=i<=N,
      1<=k<=E;

  /*****************Joves aigua terra *****/
  /*r60*/ Y{i,j}+[]'{k*10+1}--> [#]'{k*10+1} :: m{3,i}:
      g{2,i}<=j<=g{3,i},1<=i<=N, 1<=k<=E;
  /*r61*/ Y{i,j}+[]'{k*10+1}--> [Wp{i,j}]'{k*10+1} :: 1-m{3,i}:
      g{2,i}<=j<=g{3,i},1<=i<=N, 1<=k<=E;
  /********************** Adults etapa 1 aigua terra
      ****************/
  /*r62*/ Y{i,j}+[]'{k*10+1}--> [#]'{k*10+1} :: m{4,i}:
      g{3,i}<j<=g{4,i},1<=i<=N, 1<=k<=E;
  /*r63*/ Y{i,j}+[]'{k*10+1}--> [Wp{i,j}]'{k*10+1} :: 1-m{4,i}:
      g{3,i}<j<=g{4,i},1<=i<=N, 1<=k<=E;
  /****************Adults etapa 2*****/
  /*r64*/ Y{i,j}+[]'{k*10+1}--> [#]'{k*10+1} :: m{5,i}:
      g{4,i}<j<g{5,i},1<=i<=N, 1<=k<=E;
  /*r65*/ Y{i,j}+[]'{k*10+1}--> [Wp{i,j}]'{k*10+1} :: 1-m{5,i}:
      g{4,i}<j<g{5,i},1<=i<=N, 1<=k<=E;
  /************* çesperana vida *********/
  /*r66*/ Y{i,g{5,i}}+[]'{k*10+1}--> [#]'{k*10+1}::1: 1<=i<=N,
      1<=k<=E;

/****************************************/

  /*r67*/[Wp{i,j}]'{k*10+1}--> W{i,j}[]'{k*10+1} :: 1:
      1<=j<g{5,i},1<=i<=N,1<=k<=E;
  /*r68*/[Lpp{i}]'{k*10+1}--> Lpp{i}[]'{k*10+1} ::
      1:1<=i<=N,1<=k<=E;

/********************  Mortality due to floods
    module*****************************/

  /***********inundacio  inviern ****/

  /*r69*/ Lpp{i}-[]'{10*k+1} --> [Lp{i}]'{10*k+1} :: 1: 1<=i<=N,
      1<=k<=E;
  /*r70*/ W{i,j}-[]'{10*k+1} --> [W{i,j}]'{10*k+1} :: 1:
      g{2,i}<=j<=g{5,i},1<=i<=N, 1<=k<=E;


  /*r71*/ [Lp{i}]'{10*k+1} -->[#]'{10*k+1} :: mf{1,i,2}: 1<=i<=N,
      1<=k<=E;
  /*r72*/ [Lp{i}]'{10*k+1} -->Lpp{i}[]'{10*k+1} :: 1-mf{1,i,2}:
      1<=i<=N, 1<=k<=E;
```

```
  /*r73*/ [W{i,j}]'{10*k+1} --> []'{10*k+1} :: mf{2,i,2}:
      g{2,i}<=j<=g{3,i},1<=i<=N, 1<=k<=E;
  /*r74*/ [W{i,j}]'{10*k+1} --> W{i,j}[]'{10*k+1}  :: 1-mf{2,i,2}:
      g{2,i}<=j<=g{3,i},1<=i<=N, 1<=k<=E;

  /*r75*/ [W{i,j}]'{10*k+1} --> []'{10*k+1} :: mf{3,i,2}:
      g{3,i}<j<=g{4,i},1<=i<=N, 1<=k<=E;
  /*r76*/ [W{i,j}]'{10*k+1} --> W{i,j}[]'{10*k+1} :: 1-mf{3,i,2}:
      g{3,i}<j<=g{4,i},1<=i<=N, 1<=k<=E;

  /*r77*/ [W{i,j}]'{10*k+1} --> []'{10*k+1}  :: mf{4,i,2}:
      g{4,i}<j<=g{5,i},1<=i<=N, 1<=k<=E;
  /*r78*/ [W{i,j}]'{10*k+1} --> W{i,j}[]'{10*k+1}  :: 1-mf{4,i,2}:
      g{4,i}<j<=g{5,i},1<=i<=N, 1<=k<=E;


  /******************** Enviroment movement
      module*****************************/

 /*** Newt ***/
  /*r79*/ -[W{1,g{4,1}}, a{1}]'{k} --> WM{1,g{4,1},v} +[]'{k} ::
      pm{k,v}:  1<=k<=E, 1<=v<=E;
  /*r80*/ -[W{1,j}, a{1}]'{k} --> WM{1,j,k} +[]'{k} :: 1:
      1<=j<=g{5,1},j<>g{4,1}, 1<=k<=E;
  /*r81*/ -[Lpp{1}, a{1}]'{k} --> LM{1,k}+[]'{k} :: 1:  1<=k<=E;
/*** trout ***/
  /*r82*/ -[W{2,j}, a{2}]'{k} --> WM{2,j,k} +[]'{k} :: 1:
      1<=j<=g{5,2}, 1<=k<=E;
  /*r83*/ -[Lpp{2}, a{2}]'{k} --> LM{2,k}+[]'{k} :: 1: 1<=k<=E;
/****************/


  /*r84*/ WM{i,j,k} +[]'{k} -->  [X{i,j+1}]'{k} :: 1:
      1<=j<=g{5,i},1<=i<=N, 1<=k<=E;

/* he possat una mortalitat quan fan la metamorfosis , ojo
    coincideixam la de joves, no te perque **/
  /*r85*/  LM{i,k} +[]'{k} -->  []'{k} :: m{3,i}: 1<=i<=N, 1<=k<=E;
  /*r86*/  LM{i,k} +[]'{k} -->  [X{i,1}]'{k} :: (1-m{3,i}):
      1<=i<=N, 1<=k<=E;

  /*r87*/ +[W{i,j}]'{k} --> []'{k} :: 1: 1<=j<=g{5,i},1<=i<=N,
      1<=k<=E;
  /*r88*/ +[Lpp{i}]'{k} --> []'{k} :: 1: 1<=i<=N, 1<=k<=E;
  /*r89*/ +[a{i}]'{k} --> []'{k} :: 1: 1<=i<=N, 1<=k<=E;

}
def main()
{
  call newt_Ampliat(2,9,100);
}
```

Listing D.1: Fitxer `.pli`.

```
"q{1} = 4987;"        "pm{2,7} = 0;"        "I{45} = 1;"
"q{2} = 2396;"        "pm{2,8} = 0;"        "I{46} = 0;"
"q{3} = 1256;"        "pm{2,9} = 0;"        "I{47} = 0;"
"q{4} = 912;"       "pm{3,1} = 0.05;"    "I{48} = 0;"
"q{5} = 178;"       "pm{3,2} = 0.05;"    "I{49} = 1;"
"q{6} = 591;"       "pm{3,3} = 0.89999;"  "I{50} = 0;"
"q{7} = 506;"       "pm{3,4} = 0;"        "I{51} = 0;"
"q{8} = 332;"       "pm{3,5} = 0;"        "I{52} = 1;"
```

```
"q{9} = 37;"          "pm{3,6} = 0;"          "I{53} = 0;"
"q{10} = 52;"         "pm{3,7} = 0;"          "I{54} = 0;"
"q{11} = 10;"         "pm{3,8} = 0;"          "I{55} = 0;"
"q{12} = 101;"          "pm{3,9} = 0;"          "I{56} = 0;"
"q{13} = 9;"          "pm{4,1} = 0;"          "I{57} = 2;"
"q{14} = 68;"         "pm{4,2} = 0;"          "I{58} = 0;"
"q{15} = 3;"          "pm{4,3} = 0;"          "I{59} = 0;"
"q{16} = 28;"         "pm{4,4} = 1;"          "I{60} = 0;"
"q{17} = 17;"         "pm{4,5} = 0;"          "I{61} = 0;"
"q{18} = 10;"         "pm{4,6} = 0;"          "I{62} = 0;"
"q{19} = 2;"          "pm{4,7} = 0;"          "I{63} = 1;"
"q{20} = 15;"         "pm{4,8} = 0;"          "I{64} = 0;"
"q{21} = 8;"          "pm{4,9} = 0;"          "I{65} = 0;"
"q{22} = 4;"          "pm{5,1} = 0;"          "I{66} = 0;"
"q{23} = 1;"          "pm{5,2} = 0;"          "I{67} = 0;"
"q{24} = 5;"          "pm{5,3} = 0;"          "I{68} = 2;"
"q{25} = 2;"          "pm{5,4} = 0;"          "I{69} = 0;"
"q{26} = 2;"          "pm{5,5} = 1;"          "I{70} = 2;"
"qc{1} = 4;"          "pm{5,6} = 0;"          "I{71} = 0;"
"qc{2} = 2;"          "pm{5,7} = 0;"          "I{72} = 0;"
"qc{3} = 1;"          "pm{5,8} = 0;"          "I{73} = 2;"
"qc{4} = 1;"          "pm{5,9} = 0;"          "I{74} = 0;"
"qc{5} = 2;"          "pm{6,1} = 0;"          "I{75} = 0;"
"qc{6} = 3;"          "pm{6,2} = 0;"          "I{76} = 0;"
"qc{7} = 1;"          "pm{6,3} = 0;"          "I{77} = 0;"
"qc{8} = 1;"          "pm{6,4} = 0;"          "I{78} = 0;"
"qc{9} = 1;"            "pm{6,5} = 0;"          "I{79} = 0;"
"g{1,1} = 0.5;"         "pm{6,6} = 0.8;"      "I{80} = 2;"
"g{1,2} = 0.5;"       "pm{6,7} = 0.1;"      "I{81} = 0;"
"g{2,1} = 1;"         "pm{6,8} = 0;"          "I{82} = 0;"
"g{2,2} = 1;"         "pm{6,9} = 0.1;"      "I{83} = 0;"
"g{3,1} = 3;"         "pm{7,1} = 0;"          "I{84} = 2;"
"g{3,2} = 1;"         "pm{7,2} = 0;"          "I{85} = 0;"
"g{4,1} = 8;"         "pm{7,3} = 0;"          "I{86} = 0;"
"g{4,2} = 1;"         "pm{7,4} = 0;"            "I{87} = 0;"
"g{5,1} = 26;"        "pm{7,5} = 0;"          "I{88} = 0;"
"g{5,2} = 6;"         "pm{7,6} = 0;"          "I{89} = 2;"
"g{6,1} = 4;"         "pm{7,7} = 0.89999;"    "I{90} = 0;"
"g{6,2} = 2;"         "pm{7,8} = 0;"          "I{91} = 0;"
"g{7,1} = 26;"          "pm{7,9} = 0.1;"      "I{92} = 0;"
"g{7,2} = 8;"         "pm{8,1} = 0;"          "I{93} = 2;"
"g{8,1} = 1;"         "pm{8,2} = 0;"          "I{94} = 0;"
"g{8,2} = 0.05;"      "pm{8,3} = 0;"          "I{95} = 0;"
"NR{1,1} = 0;"          "pm{8,4} = 0;"          "I{96} = 2;"
"NR{1,2} = 0;"          "pm{8,5} = 0;"          "I{97} = 0;"
"NR{1,3} = 0;"          "pm{8,6} = 0;"          "I{98} = 0;"
"NR{1,4} = 0;"        "pm{8,7} = 0;"          "I{99} = 0;"
"NR{1,5} = 0;"          "pm{8,8} = 1;"          "I{100} = 1;"
"NR{1,6} = 0;"          "pm{8,9} = 0;"          "p{1} = 0.62999;"
"NR{1,7} = 0;"        "pm{9,1} = 0;"          "p{2} = 0.1;"
"NR{1,8} = 0;"          "pm{9,2} = 0;"          "pw{1,1} = 0.69999;"
"NR{1,9} = 0;"          "pm{9,3} = 0;"          "pw{2,1} = 0.69999;"
"NR{2,1} = 0;"          "pm{9,4} = 0;"          "pw{3,1} = 0.69999;"
"NR{2,2} = 400;"      "pm{9,5} = 0;"          "pw{4,1} = 0.69999;"
"NR{2,3} = 0;"          "pm{9,6} = 0;"          "pw{5,1} = 0.69999;"
"NR{2,4} = 0;"          "pm{9,7} = 0.1;"      "pw{6,1} = 0.69999;"
"NR{2,5} = 0;"          "pm{9,8} = 0;"          "pw{7,1} = 0.69999;"
"NR{2,6} = 0;"          "pm{9,9} = 0.89999;"    "pw{8,1} = 0.69999;"
"NR{2,7} = 0;"          "I{1} = 3;"             "pw{9,1} = 1;"
"NR{2,8} = 0;"          "I{2} = 0;"             "pw{10,1} = 1;"
"NR{2,9} = 0;"          "I{3} = 0;"               "pw{11,1} = 1;"
"m{1,1} = 0.625;"     "I{4} = 0;"             "pw{12,1} = 1;"
```

```
"m{1,2} = 0.05;"     "I{5} = 0;"          "pw{13,1} = 1;"
"m{2,1} = 0.08299;"  "I{6} = 1;"          "pw{14,1} = 1;"
"m{2,2} = 0.05;"     "I{7} = 0;"          "pw{15,1} = 1;"
"m{3,1} = 0.07999;"  "I{8} = 1;"          "pw{16,1} = 1;"
"m{3,2} = 0.05;"     "I{9} = 0;"          "pw{17,1} = 1;"
"m{4,1} = 0.05;"     "I{10} = 0;"         "pw{18,1} = 1;"
"m{4,2} = 0.05;"     "I{11} = 1;"         "pw{19,1} = 1;"
"m{5,1} = 0.1;"      "I{12} = 0;"         "pw{20,1} = 1;"
"m{5,2} = 0.05;"     "I{13} = 2;"         "pw{21,1} = 1;"
"mf{1,1,1} = 0.97;"  "I{14} = 0;"         "pw{22,1} = 1;"
"mf{1,2,1} = 0.5;"   "I{15} = 2;"         "pw{23,1} = 1;"
"mf{2,1,1} = 0.3;"   "I{16} = 0;"         "pw{24,1} = 1;"
"mf{2,2,1} = 0.5;"   "I{17} = 0;"         "pw{25,1} = 1;"
"mf{3,1,1} = 0.64999;" "I{18} = 0;"       "pw{26,1} = 1;"
"mf{3,2,1} = 0.5;"   "I{19} = 0;"         "pw{1,2} = 1;"
"mf{4,1,1} = 0.64999;" "I{20} = 0;"       "pw{2,2} = 1;"
"mf{4,2,1} = 0.5;"   "I{21} = 0;"         "pw{3,2} = 1;"
"mf{1,1,2} = 0.97;"  "I{22} = 2;"         "pw{4,2} = 1;"
"mf{1,2,2} = 0.5;"   "I{23} = 2;"         "pw{5,2} = 1;"
"mf{2,1,2} = 0.3;"   "I{24} = 0;"         "pw{6,2} = 1;"
"mf{2,2,2} = 0.5;"   "I{25} = 0;"         "pw{7,2} = 1;"
"mf{3,1,2} = 0.64999;" "I{26} = 0;"       "pw{8,2} = 1;"
"mf{3,2,2} = 0.5;"   "I{27} = 0;"         "pw{9,2} = 1;"
"mf{4,1,2} = 0.64999;" "I{28} = 1;"       "pw{10,2} = 1;"
"mf{4,2,2} = 0.5;"   "I{29} = 0;"         "pw{11,2} = 1;"
"pm{1,1} = 0.87;"    "I{30} = 0;"         "pw{12,2} = 1;"
"pm{1,2} = 0.02999;" "I{31} = 0;"         "pw{13,2} = 1;"
"pm{1,3} = 0.1;"     "I{32} = 0;"         "pw{14,2} = 1;"
"pm{1,4} = 0;"       "I{33} = 0;"         "pw{15,2} = 1;"
"pm{1,5} = 0;"       "I{34} = 0;"         "pw{16,2} = 1;"
"pm{1,6} = 0;"       "I{35} = 2;"         "pw{17,2} = 1;"
"pm{1,7} = 0;"       "I{36} = 0;"         "pw{18,2} = 1;"
"pm{1,8} = 0;"       "I{37} = 2;"         "pw{19,2} = 1;"
"pm{1,9} = 0;"       "I{38} = 0;"         "pw{20,2} = 1;"
"pm{2,1} = 0.02999;" "I{39} = 0;"         "pw{21,2} = 1;"
"pm{2,2} = 0.87;"    "I{40} = 0;"         "pw{22,2} = 1;"
"pm{2,3} = 0.1;"     "I{41} = 2;"         "pw{23,2} = 1;"
"pm{2,4} = 0;"       "I{42} = 0;"         "pw{24,2} = 1;"
"pm{2,5} = 0;"       "I{43} = 0;"         "pw{25,2} = 1;"
"pm{2,6} = 0;"       "I{44} = 0;"         "pw{26,2} = 1;"
```

Listing D.2: Fitxer `.var`.

```
m{3,1}=0.06:0.09:0.01
m{4,1}=0.06:0.09:0.01
m{5,1}=0.06:0.09:0.01
p{1}=0.55:0.65:0.05
p{2}=0.07:0.13:0.02
```

Listing D.3: Fitxer `.cal`.

```
X{1}
```

Listing D.4: Fitxer `.esp`.

```
1:X{1}*17234.68
2:X{1}*27605.94
3:X{1}*49427.16
4:X{1}*80226.33
5:X{1}*110597.49
6:X{1}*64666.66
```

```
7:X{1}*99367.48
8:X{1}*63611.38
9:X{1}*102828.61
10:X{1}*142374.86
11:X{1}*76982.72
12:X{1}*117348.16
13:X{1}*64477.88
14:X{1}*98945.26
15:X{1}*56257.02
16:X{1}*97437.37
17:X{1}*147028.78
18:X{1}*191039.15
19:X{1}*204307.16
```

Listing D.5: Fitxer `.avg`.

Apèndix E

Llibreries MPI per a Python

Com a part del procés de desenvolupament d'aquesta aplicació, s'han realitzat estudis sobre tecnologies i llibreries que es troben disponibles per al seu ús. Un d'aquests casos és la llibreria per a la utilització de MPI en aplicacions desenvolupades amb Python enlloc de amb C.

MPI és un entorn de pas de missatges que ha estat desenvolupat amb el llenguatge C, per tant disposa d'un ampli ús en aplicacions desenvolupades tant en C com C++. De totes formes, existeixen llibreries per a altres llenguatges de programació que, o bé realitzen una implementació completament pròpia en aquest llenguatge o bé fan de passarel·la entre aquest llenguatge i la llibreria en C.

El cas de les llibreries de MPI per a Python és el de les que fan de passarel·la entre el llenguatge hoste i les llibreries natives en C. Aprofiten aquest fet ja que Python és un llenguatge interpretat que permet fer crides directes a C.

Hem estudiat un conjunt de 6 llibreries disponibles per a Python. Per cada llibreria hem estudiat la seva completesa (si implementa totes les funcions de MPI o només un subconjunt), la disponibilitat de suport i la última actualització realitzada. També hem intentat fer-nos una idea de si el seu ús es troba força estes o no.

En la taula E.1 trobem el resum dels resultats obtinguts:

Taula E.1: Taula de llibreries MPI per a Python.

Llibreria	Completesa	Suport	Antiguitat	Ús
mpi4py	Si[1]	Excel·lent	Recent	Alt
pyMPI	Subconjunt	Desatès	Vell	Moderat
pypar	Subconjunt	Correcte	Mitjà	Alt
SciPy	Subconjunt	Desatès	Vell	Baix
boost.MPI	Subconjunt	Correcte	Mitjà	Baix
MYMPI	Subconjunt	Desatès	Vell	Molt baix

[1] Només manca la implementació d'una operació que es pot realitzar com la concatenació de dues altres.